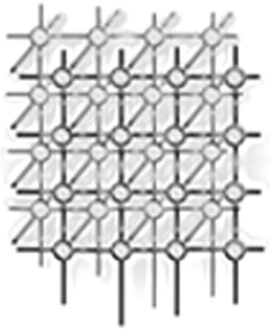
CONCURRENCY—PRACTICE AND EXPERIENCE
*Concurrency: Pract. Exper.* 2002; **00**:1–7

# Peer-to-Peer Grid Databases for Web Service Discovery



Wolfgang Hoschek

*CERN IT Division*
*European Organization for Nuclear Research*
*1211 Geneva 23, Switzerland,* wolfgang.hoschek@cern.ch

**SUMMARY**

**Grids are collaborative distributed Internet systems characterized by large scale, heterogeneity, lack of central control, multiple autonomous administrative domains, unreliable components and frequent dynamic change. In such systems, it is desirable to maintain and query dynamic and timely information about active participants such as services, resources and user communities. The *web services* vision promises that programs are made more flexible, adaptive and powerful by querying Internet databases (registries) at runtime in order to discover information and network attached building blocks, enabling the assembly of distributed higher-level components.**

**In support of this vision, we introduce the *Web Service Discovery Architecture (WSDA)*, which subsumes an array of disparate concepts, interfaces and protocols under a single semi-transparent umbrella. WSDA specifies a small set of orthogonal multi-purpose communication primitives (building blocks) for discovery, covering service identification, service description retrieval, data publication as well as minimal and powerful query support. The individual primitives can be combined and plugged together by specific clients and services to yield a wide range of behaviors and emerging synergies. Based on WSDA, we introduce the *hyper registry*, which is a centralized database node for discovery of dynamic distributed content. It supports XQueries over a tuple set from a dynamic XML data model. We address the problem of maintaining dynamic and timely information populated from a large variety of unreliable, frequently changing, autonomous and heterogeneous remote data sources.**

**However, in a large cross-organizational system, the set of information tuples is partitioned over many such distributed nodes, for reasons including autonomy, scalability, availability, performance and security. This suggests the use of Peer-to-Peer (P2P) query technology. Consequently, we propose the WSDA based *Unified Peer-to-Peer Database Framework (UPDF)* and its corresponding *Peer Database Protocol (PDP)*. They are unified in the sense that they allow to express specific discovery applications for a wide range of data types, node topologies (e.g. ring, tree, graph), query languages (e.g. XQuery, SQL), query response modes (e.g. Routed, Direct and Referral Response), neighbor selection policies, pipelining, timeout and scope policies.**

**We describe the first steps towards the convergence of Grid Computing, Peer-to-Peer Computing, Distributed Databases and Web Services. The uniformity and wide applicability of our approach is distinguished from related work, which (1) addresses some but not all problems, and (2) does not propose a unified framework.**

KEY WORDS: Grid Database; Peer-to-Peer Network; Web Service Architecture; Service Discovery

## 1. Introduction

The fundamental value proposition of computer systems has long been their potential to automate well-defined repetitive tasks. With the advent of distributed computing, the Internet and WWW technologies in particular, the focus has been broadened. Increasingly, computer systems are seen as enabling tools for effective long distance communication and collaboration. Colleagues (and programs) with shared interests can better work together, with less respect to the physical location of themselves and required devices and machinery. The traditional departmental team is complemented by cross-organizational virtual teams, operating in an open, transparent manner. Such teams have been termed *virtual organizations* [1]. This opportunity to further extend knowledge appears natural to science communities since they have a deep tradition in drawing their strength from stimulating partnerships across administrative boundaries. In particular, Grid Computing, Peer-to-Peer Computing, Distributed Databases and Web Services introduce core concepts and technologies for *Making the Global Infrastructure a Reality*. Let us look at these in more detail.

***Grids.*** Grid technology attempts to support flexible, secure, coordinated information sharing among dynamic collections of individuals, institutions and resources. This includes data sharing but also includes access to computers, software and devices required by computation and data-rich collaborative problem solving [1]. These and other advances of distributed computing are necessary to increasingly make it possible to join loosely coupled people and resources from multiple organizations. Grids are collaborative distributed Internet systems characterized by large scale, heterogeneity, lack of central control, multiple autonomous administrative domains, unreliable components and frequent dynamic change.

For example, the scale of the next generation Large Hadron Collider project at CERN, the European Organization for Nuclear Research, motivated the construction of the European Data Grid (EDG) [2], which is a global software infrastructure that ties together a massive set of people and computing resources spread over hundreds of laboratories and university departments. This includes thousands of network services, tens of thousands of CPUs, WAN Gigabit networking as well as Petabytes of disk and tape storage [3]. Many entities can now collaborate among each other to enable the analysis of High Energy Physics (HEP) experimental data: the HEP user community and its multitude of institutions, storage providers, as well as network, application and compute cycle providers. Users utilize the services of a set of remote application providers to submit jobs, which in turn are executed by the services of compute cycle providers, using storage and network provider services for I/O. The services necessary to execute a given task often do not reside in the same administrative domain. Collaborations may have a rather static configuration, or they may be more dynamic and fluid, with users and service providers joining and leaving frequently, and configurations as well as usage policies often changing.

***Services.*** Component oriented software development has advanced to a state where a large fraction of the functionality required for typical applications is available through third party libraries, frameworks and tools. These components are often reliable, well documented and maintained, and designed with the intention to be reused and customized. For many software developers the key skill is no longer hard core programming, but rather the ability to find, assess and integrate building blocks from a large variety of third parties.

The software industry has steadily moved towards more software execution flexibility. For example, dynamic linking allows for easier customization and upgrade of applications than static linking. Modern programming languages such as Java use an even more flexible link model that delays linking until the last possible moment (the time of method invocation). Still, most software expects to link and run against third party functionality installed on the local computer executing the program. For example, a word processor is locally installed together with all its internal building blocks such as spell checker, translator, thesaurus and modules for import and export of various data formats. The network is not an integral part of the software execution model whereas the local disk and operating system certainly are.

The maturing of Internet technologies has brought increased ease-of-use and abstraction through higher-level protocol stacks, improved APIs, more modular and reusable server frameworks and correspondingly powerful tools. The way is now paved for the next step towards increased software execution flexibility. In this scenario, some components are network-attached and made available in the form of network *services* for use by the general public, collaborators or commercial customers. Internet Service Providers (ISPs) offer to run and maintain reliable services on behalf of clients through hosting environments. Rather than invoking functions of a local library, the application now invokes functions on remote components, in the ideal case to the same effect. Examples of a service are:

- A replica catalog implementing an interface that, given an identifier (logical file name), returns the global storage locations of replicas of the specified file.
- A replica manager supporting file replica creation, deletion and management as well as remote shutdown and change notification via publish/subscribe interfaces.
- A storage service offering GridFTP transfer, an explicit TCP buffer size tuning interface as well as administration interfaces for management of files on local storage systems. An auxiliary interface supports queries over access logs and statistics kept in a registry that is deployed on a centralized high availability server, and shared by multiple such storage services of a computing cluster.
- A gene sequencing, language translation or an instant news and messaging service.

Remote invocation is always necessary for some demanding applications that cannot (exclusively) be run locally on the computer of a user because they depend on a set of resources scattered over multiple remote domains. Examples include computationally demanding gene sequencing, business forecasting, climate change simulation and astronomical sky surveying as well as data-intensive High Energy Physics analysis sweeping over Terabytes of data. Such applications can reasonably only be run on a remote supercomputer or several large computing

clusters with massive CPU, network, disk and tape capacities, as well as an appropriate software environment matching minimum standards.

The most straightforward but also most inflexible configuration approach is to hard wire the location, interface, behavior and other properties of remote services into the local application. Loosely coupled decentralized systems call for solutions that are more flexible and can seamlessly adapt to changing conditions. For example, if a user turns out to be less than happy with the perceived quality of a word processor's remote spell checker, he/she may want to plug in another spell checker. Such dynamic plug-ability may become feasible if service implementations adhere to some common interfaces and network protocols, and if it is possible to match services against an interface and network protocol specification. An interesting question then is: *What infrastructure is necessary to enable a program to have the capability to search the Internet for alternative but similar services and dynamically substitute these?*

***Web Services.*** As communication protocols and message formats are standardized on the Internet, it becomes increasingly possible and important to be able to describe communication mechanisms in some structured way. A service description language addresses this need by defining a grammar for describing web services as collections of service interfaces capable of executing operations over network protocols to endpoints. Service descriptions provide documentation for distributed systems and serve as a recipe for automating the details involved in application communication [4]. In contrast to popular belief, a web service is neither required to carry XML messages, nor to be bound to SOAP [5] or the HTTP protocol, nor to run within a .NET hosting environment, although all of these technologies may be helpful for implementation. For clarity, service descriptions in this chapter are formulated in the Simple Web Service Description Language (SWSDL), as introduced in our prior studies [6]. SWSDL describes the interfaces of a distributed service object system. It is a compact pedagogical vehicle trading flexibility for clarity, not an attempt to replace the WSDL [4] standard. As an example, assume we have a simple scheduling service that offers an operation `submitJob` that takes a job description as argument. The function should be invoked via the HTTP protocol. A valid SWSDL service description reads as follows:

```
<service>
  <interface type = "http://gridforum.org/Scheduler-1.0">
    <operation>
      <name>void submitJob(String jobdescription)</name>
      <allow> http://cms.cern.ch/everybody </allow>
      <bind:http verb="GET" URL="https://sched.cern.ch/submitjob"/>
    </operation>
  </interface>
</service>
```

It is important to note that the concept of a service is a logical rather than a physical concept. For efficiency, a *container* of a virtual hosting environment such as the Apache Tomcat servlet container may be used to run more than one service or interface in the same process or thread. The service interfaces of a service may, but need not, be deployed on the same host. They may be spread over multiple hosts across the LAN or WAN and even span administrative domains. This notion allows speaking in an abstract manner about a coherent interface bundle

without regard to physical implementation or deployment decisions. We speak of a *distributed (local) service*, if we know and want to stress that service interfaces are indeed deployed across hosts (or on the same host). Typically, a service is persistent (long lived), but it may also be transient (short lived, temporarily instantiated for the request of a given user).

The next step towards increased execution flexibility is the (still immature and hence often hyped) *web services* vision [6, 7] of distributed computing where programs are no longer configured with static information. Rather, the promise is that programs are made more flexible, adaptive and powerful by querying Internet databases (registries) at runtime in order to discover information and network attached third-party building blocks. Services can advertise themselves and related metadata via such databases, enabling the assembly of distributed higher-level components. While advances have recently been made in the field of web service specification [4], invocation [5] and registration [8], the problem of how to use a rich and expressive general-purpose query language to discover services that offer functionality matching a detailed specification has so far received little attention. A natural question arises. *How precisely can a local application discover relevant remote services?*

For example, a data-intensive High Energy Physics analysis application looks for remote services that exhibit a suitable combination of characteristics, including appropriate interfaces, operations and network protocols as well as network load, available disk quota, access rights, and perhaps Quality of Service and monetary cost. It is thus of critical importance to develop capabilities for rich service discovery as well as a query language that can support advanced resource brokering. What is more, it is often necessary to use several services in combination to implement the operations of a request. For example, a request may involve the combined use of a file transfer service (to stage input and output data from remote sites), a replica catalog service (to locate an input file replica with good data locality), a request execution service (to run the analysis program) and finally again a file transfer service (to stage output data back to the user desktop). In such cases it is often helpful to consider correlations. For example, a scheduler for data-intensive requests may look for input file replica locations with a fast network path to the execution service where the request would consume the input data. If a request involves reading large amounts of input data, it may be a poor choice to use a host for execution that has poor data locality with respect to an input data source, even if it is very lightly loaded. *How can one find a set of correlated services fitting a complex pattern of requirements and preferences?*

If one instance of a service can be made available, a natural next step is to have more than one identical distributed instance, for example to improve availability and performance. Changing conditions in distributed systems include latency, bandwidth, availability, location, access rights, monetary cost and personal preferences. For example, adaptive users or programs may want to choose a particular instance of a content download service depending on estimated download bandwidth. If bandwidth is degraded in the middle of a download, a user may want to switch transparently to another download service and continue where he/she left off. *On what basis could one discriminate between several instances of the same service?*

***Databases.*** In a large heterogeneous distributed system spanning multiple administrative domains, it is desirable to maintain and query dynamic and timely information about the active participants such as services, resources and user communities. Examples are a (world-wide)

service discovery infrastructure for a Data Grid, the Domain Name System (DNS), the email infrastructure, the World Wide Web, a monitoring infrastructure or an instant news service. The shared information may also include Quality of Service description, files, current network load, host information, stock quotes, etc. However, the set of information tuples in the universe is partitioned over one or more database nodes from a wide range of system topologies, for reasons including autonomy, scalability, availability, performance and security. As in a data integration system [9, 10, 11], the goal is to exploit several independent information sources as if they were a single source. This enables queries for information, resource and service discovery and collective collaborative functionality that operate on the system as a whole, rather than on a given part of it. For example, it allows a search for descriptions of services of a file sharing system, to determine its total download capacity, the names of all participating organizations, etc.

However, in such large distributed systems it is hard to keep track of metadata describing participants such as services, resources, user communities and data sources. Predictable, timely, consistent and reliable global state maintenance is infeasible. The information to be aggregated and integrated may be outdated, inconsistent, or not available at all. Failure, misbehavior, security restrictions and continuous change are the norm rather than the exception. The problem of how to support expressive general-purpose discovery queries over a view that integrates autonomous dynamic database nodes from a wide range of distributed system topologies has so far not been addressed. Consider an instant news service that aggregates news from a large variety of autonomous remote data sources residing within multiple administrative domains. New data sources are being integrated frequently and obsolete ones are dropped. One cannot force control over multiple administrative domains. Reconfiguration or physical moving of a data source is the norm rather than the exception. The question then is: *How can one keep track of and query the metadata describing the participants of large cross-organizational distributed systems undergoing frequent change?*

***Peer-to-Peer Networks.*** It is not obvious how to enable powerful discovery query support and collective collaborative functionality that operate on the distributed system as a whole, rather than on a given part of it. Further, it is not obvious how to allow for search results that are fresh, allowing time-sensitive dynamic content. Distributed (relational) database systems [12] assume tight and consistent central control and hence are infeasable in Grid environments, which are characterized by heterogeneity, scale, lack of central control, multiple autonomous administrative domains, unreliable components and frequent dynamic change. It appears that a Peer-to-Peer (P2P) database network may be well suited to support dynamic distributed database search, for example for service discovery.

In systems such as Gnutella [13], Freenet [14], Tapestry [15], Chord [16] and Globe [17], the overall P2P idea is as follows. Rather than have a centralized database, a distributed framework is used where there exist one or more autonomous database nodes, each maintaining its own, potentially heterogeneous, data. Queries are no longer posed to a central database; instead, they are recursively propagated over the network to some or all database nodes, and results are collected and send back to the client. A node holds a set of tuples in its database. Nodes are interconnected with links in any arbitrary way. A link enables a node to query another node. A *link topology* describes the link structure among nodes. The centralized model has a single node

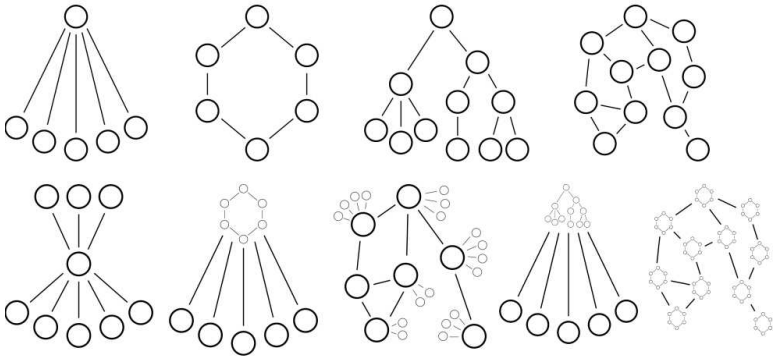


Figure 1. Example Link Topologies [18].

only. For example, in a service discovery system, a link topology can tie together a distributed set of administrative domains, each hosting a registry node holding descriptions of services local to the domain. Several link topology models covering the spectrum from centralized models to fine-grained fully distributed models can be envisaged, among them single node, star, ring, tree, graph and hybrid models [18]. Figure 1 depicts some example topologies.

In any kind of P2P network, nodes may publish themselves to other nodes, thereby forming a topology. In a P2P network for service discovery, a *node* is a service that exposes *at least* interfaces for publication and P2P queries. Here, nodes, services and other content providers may publish (their) service descriptions and/or other metadata to one or more nodes. Publication enables distributed node topology construction (e.g. ring, tree or graph) and at the same time constructs the federated database searchable by queries. In other examples, nodes may support replica location [19], replica management and optimization [20, 21], interoperable access to grid-enabled relational databases [22], gene sequencing or multi-lingual translation, actively using the network to discover services such as replica catalogs, remote gene mappers or language dictionaries.

***Organization of this Chapter.*** This chapter distills and generalizes the essential properties of the discovery problem and then develops solutions that apply to a wide range of large distributed Internet systems. It shows how to support expressive general-purpose queries over a view that integrates autonomous dynamic database nodes from a wide range of distributed system topologies. We describe the first steps towards the convergence of Grid Computing, Peer-to-Peer Computing, Distributed Databases and Web Services. The remainder of this chapter is organized as follows:

Section 2 addresses the problems of maintaining dynamic and timely information populated from a large variety of unreliable, frequently changing, autonomous and heterogeneous remote data sources. We design a database for XQueries over dynamic distributed content – the so-called *hyper registry*.

Section 3 defines the *Web Service Discovery Architecture (WSDA)*, which views the Internet as a large set of services with an extensible set of well-defined interfaces. It specifies a small set of orthogonal multi-purpose communication primitives (building blocks) for discovery. These primitives cover service identification, service description retrieval, data publication as well as minimal and powerful query support. WSDA promotes interoperability, embraces industry standards, and is open, modular, unified and simple yet powerful.

Sections 4 and 5 describe the *Unified Peer-to-Peer Database Framework (UPDF)* and corresponding *Peer Database Protocol (PDP)* for general-purpose query support in large heterogeneous distributed systems spanning many administrative domains. They are unified in the sense that they allow to express specific discovery applications for a wide range of data types, node topologies, query languages, query response modes, neighbor selection policies, pipelining characteristics, timeout and other scope options.

Section 6 discusses related work. Finally, Section 7 summarizes and concludes this chapter. We also outline interesting directions for future research.

## 2. A Database for Discovery of Distributed Content

In a large distributed system, a variety of information describes the state of autonomous entities from multiple administrative domains. Participants frequently join, leave and act on a best effort basis. Predictable, timely, consistent and reliable global state maintenance is infeasible. The information to be aggregated and integrated may be outdated, inconsistent, or not available at all. Failure, misbehavior, security restrictions and continuous change are the norm rather than the exception. The key problem then is:

- *How should a database node maintain information populated from a large variety of unreliable, frequently changing, autonomous and heterogeneous remote data sources? In particular, how should it do so without sacrificing reliability, predictability and simplicity? How can powerful queries be expressed over time-sensitive dynamic information?*

A type of database is developed that addresses the problem. A database for XQueries over dynamic distributed content is designed and specified – the so-called *hyper registry*. The registry has a number of key properties. An XML data model allows for structured and semi-structured data, which is important for integration of heterogeneous content. The XQuery language [23] allows for powerful searching, which is critical for non-trivial applications. Database state maintenance is based on soft state, which enables reliable, predictable and simple content integration from a large number of autonomous distributed content providers. Content link, content cache and a hybrid pull/push communication model allow for a wide range of dynamic content freshness policies, which may be driven by all three system components: content provider, registry and client.

A hyper registry has a database that holds a set of tuples. A *tuple* may contain a piece of arbitrary *content*. Examples of content include a service description expressed in WSDL [4], a Quality of Service description, a file, file replica location, current network load, host information, stock quotes, etc. A tuple is annotated with a *content link* pointing to the authoritative data source of the embedded content.

### 2.1. Content Link and Content Provider

***Content Link.*** A *content link* may be any arbitrary URI. However, most commonly, it is an HTTP(S) URL, in which case it points to the content of a content provider, and an HTTP(S) GET request to the link must return the current (up-to-date) content. In other words, a simple hyperlink is employed. In the context of service discovery, we use the term *service link* to denote a content link that points to a service description. Content links can freely be chosen as long as they conform to the URI and HTTP URL specification [24]. Examples of content links are:

```
urn:/iana/dns/ch/cern/cn/techdoc/94/1642-3
urn:uuid:f81d4fae-7dec-11d0-a765-00a0c91e6bf6
http://sched.cern.ch:8080/getServiceDescription.wsdl
https://cms.cern.ch/getServiceDesc?id=4712&cache=disable
http://phone.cern.ch/lookup?query="select phone from book where phone=4711"
http://repcat.cern.ch/getPFNs?lfn="myLogicalFileName"
```

***Content Provider.*** A *content provider* offers information conforming to a homogeneous global data model. In order to do so, it typically uses some kind of internal mediator to transform information from a local or proprietary data model to the global data model. A content provider can be seen as a gateway to heterogeneous content sources. A content provider is an umbrella term for two components, namely a presenter and a publisher. The *presenter* is a service and answers HTTP(S) GET content retrieval requests from a registry or client (subject to local security policy). The *publisher* is a piece of code that publishes content link, and perhaps also content, to a registry. The publisher need not be a service, although it uses HTTP(S) POST for transport of communications. The structure of a content provider and its interaction with a registry and a client are depicted in Figure 2 (a). Note that a client can bypass a registry and directly pull current content from a provider. Figure 2 (b) illustrates a registry with several content providers and clients.

Just as in the dynamic WWW that allows for a broad variety of implementations for the given protocol, it is left unspecified how a presenter computes content on retrieval. Content can be static or dynamic (generated on the fly). For example, a presenter may serve the content directly from a file or database, or from a potentially outdated cache. For increased accuracy, it may also dynamically recompute the content on each request. Consider the example providers in Figure 3. A simple but nonetheless very useful content provider uses a commodity HTTP server such as Apache to present XML content from the file system. A simple `cron` job monitors the health of the Apache server and publishes the current state to a registry. Another example of a content provider is a Java servlet that makes available data kept in a relational or LDAP database system. A content provider can execute legacy command line tools to publish system state information such as network statistics, operating system and type of CPU. Another example of a content provider is a network service such as a replica catalog that (in addition to servicing replica lookup requests) publishes its service description and/or link so that clients may discover and subsequently invoke it.

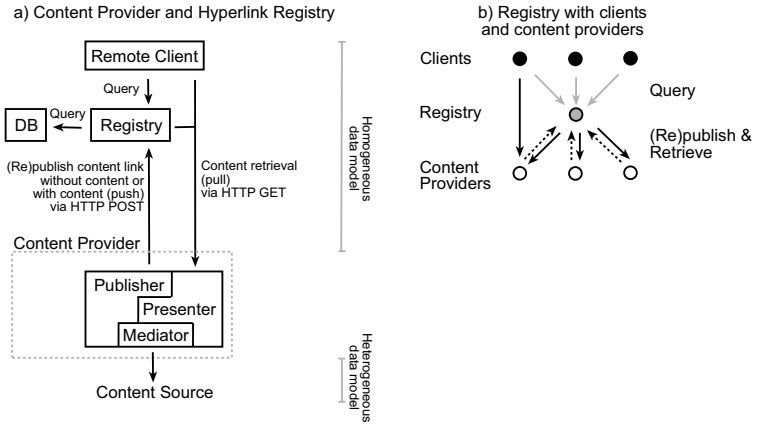



Figure 2. Content Provider and Hyper Registry.

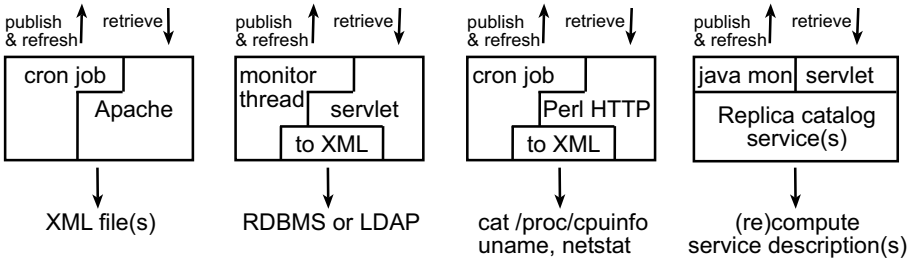



Figure 3. Example Content Providers.

## 2.2. Publication

In a given context, a content provider can publish content of a given type to one or more registries. More precisely, a content provider can publish a dynamic pointer called a content link, which in turn enables the registry (and third parties) to retrieve the current (up-to-date) content. For efficiency, the `publish` operation takes as input a set of zero or more tuples. In what we propose to call the *Dynamic Data Model (DDM)*, each XML tuple has a content link, a type, a context, four soft state time stamps, and (optionally) metadata and content. A tuple is an annotated multi-purpose soft state data container that may contain a piece of arbitrary *content* and allows for refresh of that content at any time, as depicted in Figure 4 and 5.

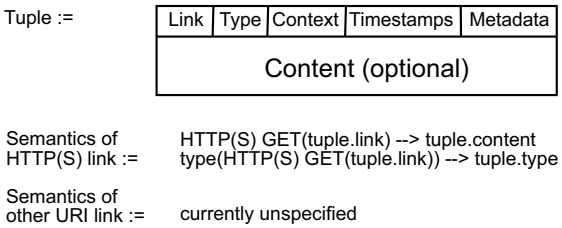



Figure 4. Tuple is an annotated multi-purpose soft state data container, and allows for dynamic refresh.

- **Link.** The content link is an URI in general, as introduced above. If it is an HTTP(S) URL, then the the current (up-to-date) content of a content provider can be retrieved (pulled) at any time.

- **Type.** The type describes *what* kind of content is being published (e.g. `service`, `application/octet-stream`, `image/jpeg`, `networkLoad`, `hostinfo`).

- **Context.** The context describes *why* the content is being published or *how* it should be used (e.g. `child, parent, x-ireferral, gnutella, monitoring`). Context and type allow a query to differente on crucial attributes even if content caching is not supported or not authorized.

- **Timestamps TS1, TS2, TS3, TC.** Based on embedded soft state time stamps defining lifetime properties, a tuple may eventually be discarded unless refreshed by a stream of timely confirmation notifications. The time stamps allow for a wide range of powerful caching policies, some of which are described below in Section 2.5.

- **Metadata.** The optional metadata element further describes the content and/or its retrieval beyond what can be expressed with the previous attributes. For example, the metadata may be a secure digital XML signature [25] of the content. It may describe the authoritative content provider or owner of the content. Another metadata example is a Web Service Inspection Language (WSIL) document [26] or fragment thereof, specifying additional content retrieval mechanisms beyond HTTP content link retrieval. The metadata argument is an extensibility element enabling customization and flexible evolution.

- **Content.** Given the link the current (up-to-date) content of a content provider can be retrieved (pulled) at any time. Optionally, a content provider can also include a copy of the current content as part of publication (push). Content and metadata can be structured or semi-structured data in the form of any arbitrary well-formed XML

```
<tupleset>
    <tuple link="http://registry.cern.ch/getDescription" type="service" ctx="parent"
           TS1="10" TC="15" TS2="20" TS3="30">
       <content>
         <service>
           <interface type="http://cern.ch/Presenter-1.0">
             <operation>
               <name>XML getServiceDescription()</name>
               <bind:http verb="GET" URL="https://registry.cern.ch/getDesc"/>
             </operation>
           </interface>

           <interface type = "http://cern.ch/XQuery-1.0">
             <operation>
               <name> XML query(XQuery query)</name>
               <bind:beep URL="beep://registry.cern.ch:9000"/>
             </operation>
           </interface>
         </service>
       </content>

       <metadata>  <owner name="http://cms.cern.ch"/> </metadata>
    </tuple>

    <tuple link="http://repcat.cern.ch/getDesc?id=4711" type="service" ctx="child"
           TS1="30" TC="0" TS2="40" TS3="50">
    </tuple>

    <tuple link="urn:uuid:f81d4fae-11d0-a765-00a0c91e6bf6"
           type="replica" TC="65" TS1="60" TS2="70" TS3="80">
       <content>
         <replicaSet LFN="urn:/iana/dns/ch/cern/cms/higgs-file" size="10000000" type="MySQL/ISAM">
             <PFN URL="ftp://storage.cern.ch/file123" readCount="17"/>
             <PFN URL="ftp://se01.infn.it/file456"    readCount="1"/>
         </replicaSet>
       </content>
    </tuple>

    <tuple link="http://monitor.cern.ch/getHosts" type="hosts" TC="65" TS1="60" TS2="70" TS3="80">
       <content>
         <hosts>
             <host name="fred01.cern.ch" os="redhat 7.2"  arch="i386" mem="512M" MHz="1000"/>
             <host name="fred02.cern.ch" os="solaris 2.7" arch="sparc" mem="8192M" MHz="400"/>
         </hosts>
       </content>
    </tuple>
</tupleset>
```

Figure 5. Example Tuple Set from Dynamic Data Model.

document or fragment[1]. An individual element may, but need not, have a schema (XML Schema [28]), in which case it must be valid according to the schema. All elements may, but need not, share a common schema. This flexibility is important for integration of heterogeneous content.

The publish operation of a registry has the signature `void publish(XML tupleset)`. Within a tuple set, a tuple is uniquely identified by its *tuple key*, which is the pair `(content link, context)`. If a key does not already exist on publication, a tuple is inserted into the registry database. An existing tuple can be updated by publishing other values under the same tuple key. An existing tuple (key) is "owned" by the content provider that created it with the first publication. It is recommended that a content provider with another identity may not be permitted to publish or update the tuple.

### 2.3. Query

Having discussed the data model and how to publish tuples, we now consider a query model. It offers two interfaces, namely `MinQuery` and `XQuery`.

***MinQuery.*** The `MinQuery` interface provides the simplest possible query support (*"select all"*-style). It returns tuples including or excluding cached content. The `getTuples()` query operation takes no arguments and returns the full set of all tuples "as is". That is, query output format and publication input format are the same (see Figure 5). If supported, output includes cached content. The `getLinks()` query operation is similar in that it also takes no arguments and returns the full set of all tuples. However, it always substitutes an empty string for cached content. In other words, the content is omitted from tuples, potentially saving substantial bandwidth. The second tuple in Figure 5 has such a form.

***XQuery.*** The `XQuery` interface provides powerful XQuery [23] support, which is important for realistic service and resource discovery use cases. XQuery is the standard XML query language developed under the auspices of the W3C. It allows for powerful searching, which is critical for non-trivial applications. Everything that can be expressed with SQL can also be expressed with XQuery. However, XQuery is a more expressive language than SQL, for example, because it supports path expressions for hierarchical navigation. Example XQueries for service discovery are depicted in Figure 6. A detailed discussion of a wide range of simple, medium and complex discovery queries and their representation in the XQuery [23] language is given in [6]. XQuery can dynamically integrate external data sources via the `document(URL)` function, which can be used to process the XML results of remote operations invoked over HTTP. For example, given a service description with a `getPhysicalFileNames(LogicalFileName)` operation, a query can match on values

[1]For clarity of exposition, the content is an XML element. In the general case (allowing non-text based content types such as `image/jpeg`), the content is a MIME [27] object. The XML based publication input tuple set and query result tuple set is augmented with an additional MIME multipart object, which is a list containing all content. The content element of a tuple is interpreted as an index into the MIME multipart object.

- *Find all (available) services.*

```
RETURN /tupleset/tuple[@type="service"]
```

- *Find all services that implement a replica catalog service interface that CMS members are allowed to use, and that have an HTTP binding for the replica catalog operation "XML getPFNs(String LFN).*

```
LET $repcat := "http://cern.ch/ReplicaCatalog-1.0"
FOR $tuple in /tupleset/tuple[@type="service"]
LET $s := $tuple/content/service
WHERE SOME $op IN $s/interface[@type = $repcat]/operation SATISFIES
  $op/name="XML getPFNs(String LFN)" AND $op/bindhttp/@verb="GET" AND contains($op/allow, "cms.cern.ch")
RETURN $tuple
```

- *Find all replica catalogs and return their physical file names (PFNs) for a given logical file name (LFN); suppress PFNs not starting with "ftp://".*

```
LET $repcat := "http://cern.ch/ReplicaCatalog-1.0"
LET $s := /tupleset/tuple[@type="service"]/content/service[interface@type = $repcat]
RETURN
  FOR $pfn IN invoke($s, $repcat, "XML getPFNs(String LFN)", "http://myhost.cern.ch/myFile")/tupleset/PFN
  WHERE starts-with($pfn, "ftp://")
  RETURN $pfn
```

- *Return the number of replica catalog services.*

```
RETURN count(/tupleset/tuple/content/service[interface/@type="http://cern.ch/ReplicaCatalog-1.0"])
```

- *Find all (execution service, storage service) pairs where both services of a pair live within the same domain. (Job wants to read and write locally).*

```
LET $executorType := "http://cern.ch/executor-1.0"
LET $storageType := "http://cern.ch/storage-1.0"
FOR $executor IN /tupleset/tuple[content/service/interface/@type=$executorType],
    $storage  IN /tupleset/tuple[content/service/interface/@type=$storageType
                                 AND domainName(@link) = domainName($executor/@link)]
RETURN <pair> {$executor} {$storage} </pair>
```

Figure 6. Example XQueries for Service Discovery.

dynamically produced by that operation. The same rules that apply to minimalist queries also apply to XQuery support. An implementation can use a modular and simple XQuery processor such as `Quip` for the operation `XML query(XQuery query)`. Because not only content, but also content link, context, type, time stamps, metadata etc. are part of a tuple, a query can also select on this information.

### 2.4. Caching

Content *caching* is important for client efficiency. The registry may not only keep content links but also a copy of the current content pointed to by the link. With caching, clients no longer need to establish a network connection for each content link in a query result set in order to obtain content. This avoids prohibitive latency, in particular in the presence of large result sets. A registry may (but need not) support caching. A registry that does not support caching ignores any content handed from a content provider. It keeps content links only. Instead of cached content it returns empty strings (see the second tuple in Figure 5 for an example). Cache coherency issues arise. The query operations of a caching registry may return tuples with stale content, i.e. content that is out of date with respect to its master copy at the content provider.

A caching registry may implement a *strong* or *weak cache coherency policy*. A strong cache coherency policy is *server invalidation* [29]. Here a content provider notifies the registry with a publication tuple whenever it has locally modified the content. We use this approach in an adapted version where a caching registry can operate according to the client push pattern (*push registry*) or server pull pattern (*pull registry*) or a hybrid thereof. The respective interactions are as follows:

- **Pull Registry.** A content provider publishes a content link. The registry then pulls the current content via content link retrieval into the cache. Whenever the content provider modifies the content, it notifies the registry with a publication tuple carrying the time the content was last modified. The registry may then decide to pull the current content again, in order to update the cache. It is up to the registry to decide if and when to pull content. A registry may pull content at any time. For example, it may dynamically pull fresh content for tuples affected by a query. This is important for frequently changing dynamic data such as network load.

- **Push Registry.** A publication tuple pushed from a content provider to the registry contains not only a content link but also its current content. Whenever a content provider modifies content, it pushes a tuple with the new content to the registry, which may update the cache accordingly.

- **Hybrid Registry.** A hybrid registry implements both pull and push interactions. If a content provider merely notifies that its content has changed, the registry may choose to pull the current content into the cache. If a content provider pushes content, the cache may be updated with the pushed content. This is the type of registry subsequently assumed whenever a caching registry is discussed.

A non-caching registry ignores content elements, if present. A publication is said to be *without content* if the content is not provided at all in the tuple. Otherwise, it is said to be *with content*. Publication without content implies that no statement at all about cached content is being made (neutral). It does *not* imply that content should not be cached or invalidated.

### 2.5. Soft State

For reliable, predictable and simple distributed state maintenance, a registry tuple is maintained as *soft state*. A tuple may eventually be discarded unless refreshed by a stream of timely confirmation notifications from a content provider. To this end, a tuple carries timestamps. A tuple is expired and removed unless explicitly renewed via timely periodic publication, henceforth termed *refresh*. In other words, a refresh allows a content provider to cause a content link and/or cached content to remain present for a further time.

The strong cache coherency policy *server invalidation* is extended. For flexibility and expressiveness, the ideas of the Grid Notification Framework [30] are adapted. The publication operation takes four absolute time stamps `TS1, TS2, TS3, TC` per tuple. The semantics are as follows. The content provider asserts that its content was last modified at time `TS1` and that its current content is expected to be valid from time `TS1` until at least time `TS2`. It is expected that the content link is alive between time `TS1` and at least time `TS3`. Time stamps must obey the constraint `TS1` $\leq$ `TS2` $\leq$ `TS3`. `TS2` triggers expiration of cached content, whereas `TS3` triggers expiration of content links. Usually, `TS1` equals the time of last modification or first publication, TS2 equals `TS1` plus some minutes or hours, and `TS3` equals `TS2` plus some hours or days. For example, `TS1`, `TS2` and `TS3` can reflect publication time, 10 minutes, and 2 hours, respectively.

A tuple also carries a timestamp `TC` that indicates the time when the tuple's embedded content (not the provider's master copy of the content) was last modified, typically by an intermediary in the path between client and content provider (e.g. the registry). If a content provider publishes with content, then we usually have `TS1=TC`. `TC` must be zero-valued if the tuple contains no content. Hence, a registry not supporting caching always has `TC` set to zero. For example, a highly dynamic network load provider may publish its link without content and `TS1=TS2` to suggest that it is inappropriate to cache its content. Constants are published with content and `TS2=TS3=infinity, TS1=TC=currentTime`. Timestamp semantics can be summarized as follows:

```
TS1 = Time content provider last modified content
TC  = Time embedded tuple content was last modified (e.g. by intermediary)
TS2 = Expected time while current content at provider is at least valid
TS3 = Expected time while content link at provider is at least valid (alive)
```

Insert, update and delete of tuples occur at the timestamp-driven state transitions summarized in Figure 7. Within a tuple set, a tuple is uniquely identified by its *tuple key*, which is the pair `(content link, context)`. A tuple can be in one of three states: *unknown*, *not cached*, or *cached*. A tuple is unknown if it is not contained in the registry (i.e. its key does not exist). Otherwise, it is known. When a tuple is assigned *not cached* state, its last internal modification time `TC` is (re)set to zero and the cache is deleted, if present. For a *not cached* tuple we have `TC < TS1`. When a tuple is assigned *cached* state, the content is updated and `TC` is set to the current time. For a *cached* tuple, we have `TC` $\geq$ `TS1`.

A tuple moves from *unknown* to *cached* or *not cached* state if the provider publishes with or without content, respectively. A tuple becomes *unknown* if its content link expires (`currentTime > TS3`); the tuple is then deleted. A provider can force tuple deletion by

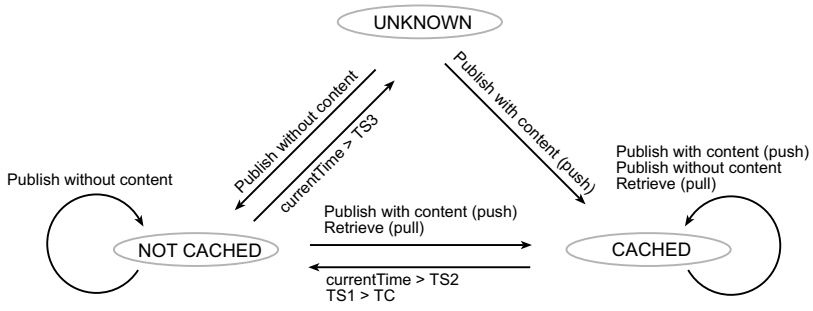



Figure 7. Soft State Transitions.

publishing with `currentTime > TS3`. A tuple is upgraded from *not cached* to *cached* state if a provider push publishes with content or if the registry pulls the current content itself via retrieval. On content pull, a registry may leave `TS2` unchanged, but it may also follow a policy that extends the lifetime of the tuple (or any other policy it sees fit). A tuple is degraded from *cached* to *not cached* state if the content expires. Such expiry occurs when no refresh is received in time (`currentTime > TS2`), or if a refresh indicates that the provider has modified the content (`TC < TS1`).

### 2.6. Flexible Freshness

Content link, content cache, a hybrid pull/push communication model and the expressive power of XQuery allow for a wide range of dynamic content freshness policies, which may be driven by all three system components: content provider, registry and client. All three components may indicate how to manage content according to their respective notions of freshness. For example, a content provider can model the freshness of its content via pushing appropriate timestamps and content. A registry can model the freshness of its content via controlled acceptance of provider publications and by actively pulling fresh content from the provider. If a result (e.g. network statistics) is up to date according to the registry, but out of date according to the client, the client can pull fresh content from providers as it sees fit. However, this is inefficient for large result sets. Nevertheless, it is important for clients that query results are returned according to their notion of freshness, in particular in the presence of frequently changing dynamic content.

Recall that it is up to the registry to decide to what extent its cache is stale, and if and when to pull fresh content. For example, a registry may implement a policy that dynamically pulls fresh content for a tuple whenever a query touches (affects) the tuple. For example, if a query interprets the content link as an identifier within a hierarchical name space (e.g. as in LDAP) and selects only tuples within a sub-tree of the name space, only these tuples should be considered for refresh.

***Refresh-on-client-demand.*** So far, a registry must guess what a client's notion of freshness might be, while at the same time maintaining its decisive authority. A client still has no way to indicate (as opposed to force) its view of the matter to a registry. We propose to address this problem with a simple and elegant *refresh-on-client-demand* strategy under control of the registry's authority. The strategy exploits the rich expressiveness and dynamic data integration capabilities of the XQuery language. The client query may itself inspect the time stamp values of the set of tuples. It may then decide itself to what extent a tuple is considered interesting yet stale. If the query decides that a given tuple is stale (e.g. if `type="networkLoad" AND TC < currentTime() - 10`), it calls the XQuery `document(URL contentLink)` function with the corresponding content link in order to pull and get handed fresh content, which it then processes in any desired way.

This mechanism makes it unnecessary for a registry to guess what a client's notion of freshness might be. It also implies that a registry does not require complex logic for query parsing, analysis, splitting, merging, etc. Moreover, the fresh results pulled by a query can be reused for subsequent queries. Since the query is executed within the registry, the registry may implement the `document` function such that it not only pulls and returns the current content, but as a side effect also updates the tuple cache in its database. A registry retains its authority in the sense that it may apply an authorization policy, or perhaps ignore the query's refresh calls altogether and return the old content instead. The refresh-on-client-demand strategy is simple, elegant and controlled. It improves efficiency by avoiding overly eager refreshes typically incurred by a guessing registry policy.

## 3. Web Service Discovery Architecture

Having defined all registry aspects in detail, we now proceed to the definition of a web service layer that promotes interoperability for Internet software. Such a layer views the Internet as a large set of services with an extensible set of well-defined interfaces. A web service consists of a set of interfaces with associated operations. Each operation may be bound to one or more network protocols and endpoints. The definition of interfaces, operations and bindings to network protocols and endpoints is given as a service description. A discovery architecture defines appropriate services, interfaces, operations and protocol bindings for discovery. The key problem is:

- *Can we define a discovery architecture that promotes interoperability, embraces industry standards, and is open, modular, flexible, unified, non-disruptive and simple yet powerful?*

We propose and specify such an architecture, the so-called *Web Service Discovery Architecture (WSDA).* WSDA subsumes an array of disparate concepts, interfaces and protocols under a single semi-transparent umbrella. It specifies a small set of orthogonal multi-purpose communication primitives (building blocks) for discovery. These primitives cover service identification, service description retrieval, data publication as well as minimal and powerful query support. The individual primitives can be combined and plugged together by specific clients and services to yield a wide range of behaviors and emerging synergies.

| Interface | Operations | Responsibility |
|---|---|---|
| `Presenter` | `XML getServiceDescription()` | Allows clients to retrieve the current description of a service and hence to bootstrap all capabilities of a service. |
| `Consumer` | `(TS4,TS5) publish(XML tupleset)` | A content provider can publish a dynamic pointer called a content link, which in turn enables the consumer (e.g. registry) to retrieve the current content. Optionally, a content provider can also include a copy of the current content as part of publication. Each input tuple has a content link, a type, a context, four time stamps, and (optionally) metadata and content. |
| `MinQuery` | `XML getTuples()`<br>`XML getLinks()` | Provides the simplest possible query support (*"select all"*-style). The `getTuples` operation returns the full set of all available tuples "as is". The minimal `getLinks` operation is identical but substitutes an empty string for cached content. |
| `XQuery` | `XML query(XQuery query)` | Provides powerful XQuery support. Executes an XQuery over the available tuple set. Because not only content, but also content link, context, type, time stamps, metadata etc. are part of a tuple, a query can also select on this information. |

Table I. WSDA Interfaces and their Respective Operations.

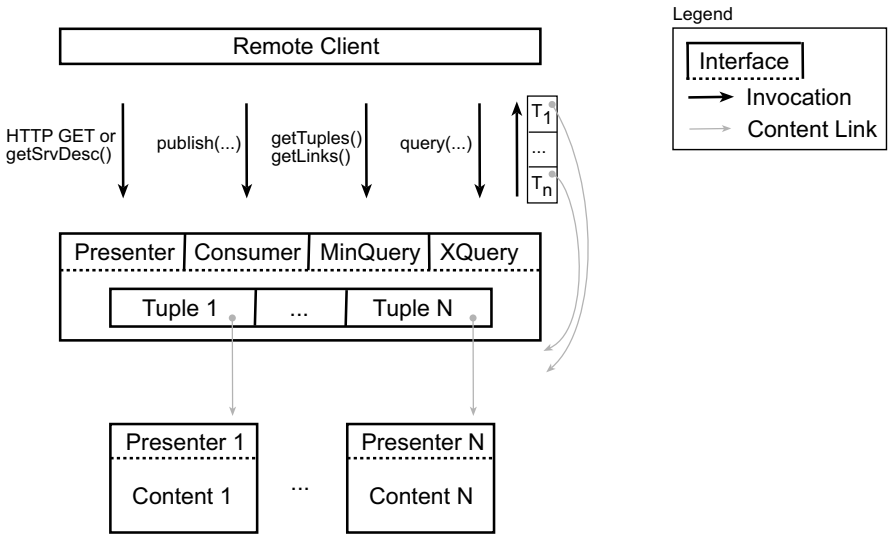



Figure 8. Interactions of Client with WSDA Interfaces.

### 3.1. Interfaces

The four WSDA interfaces and their respective operations are summarized in Table I. Figure 8 depicts the interactions of a client with implementations of these interfaces. Let us discuss the interfaces in more detail.

***Presenter.*** The `Presenter` interface allows clients to retrieve the current (up-to-date) service description. Clearly, clients from anywhere must be able to retrieve the current description of a service (subject to local security policy). Hence, a service needs to present (make available) to clients the means to retrieve the service description. To enable clients to query in a global context, some identifier for the service is needed. Further, a description retrieval mechanism is required to be associated with each such identifier. Together these are the bootstrap key (or handle) to all capabilities of a service.

In principle, identifier and retrieval mechanisms could follow any reasonable convention, suggesting the use of any arbitrary URI. In practice, however, a fundamental mechanism such as service discovery can only hope to enjoy broad acceptance, adoption and subsequent ubiquity if integration of legacy services is made easy. The introduction of service discovery as a new and additional auxiliary service capability should require as little change as possible to the large base of valuable existing legacy services, preferably no change at all. It should be possible to implement discovery-related functionality without changing the core service. Further, to help easy implementation the retrieval mechanism should have a very narrow interface and be as simple as possible.

Thus, for generality, we define that an identifier may be any URI. However, in support of the above requirements, the identifier is most commonly chosen to be a URL [24], and the retrieval mechanism is chosen to be HTTP(S). If so, we define that an HTTP(S) GET request to the identifier must return the current service description (subject to local security policy). In other words, a simple hyperlink is employed. In the remainder of this chapter, we will use the term *service link* for such an identifier enabling service description retrieval. Like in the WWW, service links (and content links, see below) can freely be chosen as long as they conform to the URI and HTTP URL specification [24].

Because service descriptions should describe the essentials of the service, it is recommended[2] that the service link concept be an integral part of the description itself. As a result, service descriptions may be retrievable via the `Presenter` interface, which defines an operation `getServiceDescription()` for this purpose. The operation is identical to service description retrieval and is hence bound to (invoked via) an HTTP(S) GET request to a given service link. Additional protocol bindings may be defined as necessary.

***Consumer.*** The `Consumer` interface allows content providers to publish a tuple set to a consumer. The publish operation has the signature `(TS4, TS5) publish(XML tupleset)`. For details, see Section 2.2.

***MinQuery.*** The `MinQuery` interface provides the simplest possible query support (*"select all"*-style). The `getTuples()` and `getLinks()` operations return tuples including and excluding cached content, respectively. For details, see Section 2.3.

Advanced query support can be expressed on top of the minimal query capabilities. Such higher-level capabilities conceptually do not belong to a consumer and minimal query interface, which are only concerned with the fundamental capability of making a content link (e.g. service

[2]In general, it is not mandatory for a service to implement any "standard" interface.

| Capability | XQuery | XPath | SQL | LDAP |
|---|---|---|---|---|
| Simple, medium and complex queries over a set of tuples | yes | no | yes | no |
| Query over structured and semi-structured data | yes | yes | no | yes |
| Query over heterogeneous data | yes | yes | no | yes |
| Query over XML data model | yes | yes | no | no |
| Navigation through hierarchical data structures (Path Expressions) | yes | yes | no | exact match only |
| Joins (combine multiple data sources into a single result) | yes | no | yes | no |
| Dynamic data integration from multiple heterog. sources such as databases, documents and remote services | yes | yes | no | no |
| Data restructuring patterns (e.g. SELECT-FROM-WHERE in SQL) | yes | no | yes | no |
| Iteration over sets (e.g. FOR clause) | yes | no | yes | no |
| General-purpose predicate expressions (WHERE clause) | yes | no | yes | no |
| Nesting several kinds of expressions with full generality | yes | no | no | no |
| Binding of variables and creating new structures from bound variables (LET clause) | yes | no | yes | no |
| Constructive queries | yes | no | no | no |
| Conditional expressions (IF ... THEN ... ELSE) | yes | no | yes | no |
| Arithmetic, comparison, logical and set expressions | yes, all | yes | yes, all | log. & string |
| Operations on data types from a type system | yes | no | yes | no |
| Quantified expressions (e.g. SOME, EVERY clause) | yes | no | yes | no |
| Standard functions for sorting, string, math, aggregation | yes | no | yes | no |
| User defined functions | yes | no | yes | no |
| Regular expression matching | yes | yes | no | no |
| Concise and easy to understand queries | yes | yes | yes | yes |

Table II. Capabilities of XQuery, XPath, SQL and LDAP query languages.

link) *reachable*[3] for clients. As an analogy, consider the related but distinct concepts of web hyper-linking and web searching: Web hyper-linking is a fundamental capability without which nothing else on the Web works. Many different kinds of web search engines using a variety of search interfaces and strategies can and are layered on top of web linking. The kind of XQuery support we propose below is certainly not the only possible and useful one. It seems unreasonable to assume that a single global standard query mechanism can satisfy all present and future needs of a wide range of communities. Multiple such mechanisms should be able to coexist. Consequently, the consumer and query interfaces are deliberately separated and kept as minimal as possible, and an additional interface type (`XQuery`) for answering XQueries is introduced.

[3] *Reachability* is interpreted in the spirit of garbage collection systems: A content link is reachable for a given client if there exists a direct or indirect retrieval path from the client to the content link.

***XQuery.*** The greater the number and heterogeneity of content and applications, the more important expressive general-purpose query capabilities become. Realistic ubiquitous service and resource discovery *stands and falls* with the ability to express queries in a rich general-purpose query language [6]. A query language suitable for service and resource discovery should meet the requirements stated in Table II (in decreasing order of significance). As can be seen from the table, LDAP, SQL and XPath do not meet a number of essential requirements, whereas the XQuery language meets all requirements and desiderata posed. The operation `XML query(XQuery query)` of the `XQuery` interface is detailed in Section 2.3.

### 3.2. Network Protocol Bindings and Services

The operations of the WSDA interfaces are bound to (carried over) a default transport protocol. The `XQuery` interface is bound to the *Peer Database Protocol (PDP)* (see Section 5). PDP supports database queries for a wide range of database architectures and response models such that the stringent demands of ubiquitous Internet discovery infrastructures in terms of scalability, efficiency, interoperability, extensibility and reliability can be met. In particular, it allows for high concurrency, low latency, pipelining as well as early and/or partial result set retrieval, both in pull and push mode. For all other operations and arguments we assume for simplicity HTTP(S) GET and POST as transport, and XML based parameters. Additional protocol bindings may be defined as necessary.

We define two kinds of example registry services: The so-called *hypermin registry* must (at least) support the three interfaces `Presenter`, `Consumer` and `MinQuery` (excluding XQuery support). A *hyper registry* must (at least) support these interfaces plus the `XQuery` interface. Put another way, any service that happens to support, among others, the respective interfaces qualifies as a hypermin registry or hyper registry. As usual, the interfaces may have endpoints that are hosted by a single container, or they may be spread across multiple hosts or administrative domains.

It is by no means a requirement that only dedicated hyper registry services and hypermin registry services may implement WSDA interfaces. Any arbitrary service may decide to offer and implement none, some or all of these four interfaces. For example, a job scheduler may decide to implement, among others, the `MinQuery` interface to indicate a simple means to discover metadata tuples related to the current status of job queues and the supported Quality of Service. The scheduler may not want to implement the `Consumer` interface because its metadata tuples are strictly read-only. Further, it may not want to implement the `XQuery` interface, because it is considered overkill for its purposes. Even though such a scheduler service does not qualify as a hypermin or hyper registry, it clearly offers useful added value. Other examples of services implementing a subset of WSDA interfaces are consumers such as an instant news service or a cluster monitor. These services may decide to implement the `Consumer` interface to invite external sources for data feeding, but they may not find it useful to offer and implement any query interface.

In a more ambitious scenario, the example job scheduler may decide to publish its local tuple set also to an (already existing) remote hyper registry service (i.e. with XQuery support). To indicate to clients how to get hold of the XQuery capability, the scheduler may simply copy the `XQuery` interface description of the remote hyper registry service and advertise it as its

own interface by including it in its own service description. This kind of *virtualization* is not a "trick", but a feature with significant practical value, because it allows for minimal implementation and maintenance effort on the part of the scheduler.

Alternatively, the scheduler may include in its local tuple set (obtainable via the `getLinks()` operation) a tuple that refers to the service description of the remote hyper registry service. An interface referral value for the context attribute of the tuple is used, as follows:

```
<tuple link="https://registry.cern.ch/getServiceDescription"
       type="service" ctx="x-ireferral://cern.ch/XQuery-1.0"
       TS1="30" TC="0" TS2="40" TS3="50">
</tuple>
```

### 3.3. Properties

WSDA has a number of key properties:

- **Standards Integration.** WSDA embraces and integrates solid and broadly accepted industry standards such as XML, XML Schema [28], the Simple Object Access Protocol (SOAP) [5], the Web Service Description Language (WSDL) [4] and XQuery [23]. It allows for integration of emerging standards such as the Web Service Inspection Language (WSIL) [26].
- **Interoperability.** WSDA promotes an interoperable web service layer on top of Internet software, because it defines appropriate services, interfaces, operations and protocol bindings. WSDA does not introduce new Internet standards. Rather, it judiciously combines existing interoperability-proven open Internet standards such as HTTP(S), URI [24], MIME [27], XML, XML Schema [28] and BEEP [31].
- **Modularity.** WSDA is modular because it defines a small set of orthogonal multi-purpose communication primitives (building blocks) for discovery. These primitives cover service identification, service description retrieval, publication, as well as minimal and powerful query support. The responsibility, definition and evolution of any given primitive is distinct and independent of that of all other primitives.
- **Ease-of-use and Ease-of-implementation.** Each communication primitive is deliberately designed to avoid any unnecessary complexity. The design principle is to *"make simple and common things easy, and powerful things possible"*. In other words, solutions are rejected that provision for powerful capabilities yet imply that even simple problems are complicated to solve. For example, service description retrieval is by default based on a simple HTTP(S) GET. Yet, we do not exclude, and indeed allow for, alternative identification and retrieval mechanisms such as the ones offered by UDDI (Universal Description, Discovery and Integration) [8], RDBMS or custom Java RMI registries (e.g. via tuple metadata specified in WSIL [26]). Further, tuple content is by default given in XML, but advanced usage of arbitrary MIME [27] content (e.g. binary images, files, MS-Word documents) is also possible. As another example, the minimal query interface requires virtually no implementation effort on the part of a client and server. Yet, where necessary, also powerful XQuery support may, but need not, be implemented and used.

- **Openness and Flexibility.** WSDA is open and flexible because each primitive can be used, implemented, customized and extended in many ways. For example, the interfaces of a service may have endpoints spread across multiple hosts or administrative domains. However, there is nothing that prevents all interfaces to be co-located on the same host or implemented by a single program. Indeed, this is often a natural deployment scenario. Further, even though default network protocol bindings are given, additional bindings may be defined as necessary. For example, an implementation of the `Consumer` interface may bind to (carry traffic over) HTTP(S), SOAP/BEEP [32], FTP or Java RMI. The tuple set returned by a query may be maintained according to a wide variety of cache coherency policies, resulting in static to highly dynamic behavior. A consumer may take any arbitrary custom action upon publication of a tuple. For example, it may interpret a tuple from a specific schema as a command or an active message, triggering tuple transformation and/or forwarding to other consumers such as loggers. For flexibility, a service maintaining a WSDA tuple set may be deployed in any arbitrary way. For example, the database can be kept in a XML file, in the same format as returned by the `getTuples` query operation. However, tuples can also be dynamically recomputed or kept in a relational database.
- **Expressive Power.** WSDA is powerful because its individual primitives can be combined and plugged together by specific clients and services to yield a wide range of behaviors. Each single primitive is of limited value all by itself. The true value of simple orthogonal multi-purpose communication primitives lies in their potential to generate powerful emerging synergies. For example, combination of WSDA primitives enables building services for replica location, name resolution, distributed auctions, instant news and messaging, software and cluster configuration management, certificate and security policy repositories, as well as Grid monitoring tools. As another example, the consumer and query interfaces can be combined to implement a Peer-to-Peer (P2P) database network for service discovery (see Section 4). Here, a node of the network is a service that exposes *at least* interfaces for publication and P2P queries.
- **Uniformity.** WSDA is unified because it subsumes an array of disparate concepts, interfaces and protocols under a single *semi-transparent* umbrella. It allows for multiple competing distributed systems concepts and implementations to coexist and to be integrated. Clients can dynamically adapt their behavior based on rich service introspection capabilities. Clearly, there exists no solution that is optimal in the presence of the heterogeneity found in real-world large cross-organizational distributed systems such as Data Grids, electronic market places and instant Internet news and messaging services. Introspection and adaption capabilities increasingly make it unnecessary to mandate a single global solution to a given problem, thereby enabling integration of collaborative systems.
- **Non-Disruptiveness.** WSDA is non-disruptive because it offers interfaces but does not mandate that every service in the universe must comply to a set of "standard" interfaces.

## 4. Peer-to-Peer Grid Databases

In a large cross-organizational system the set of information tuples is partitioned over many distributed nodes, for reasons including autonomy, scalability, availability, performance and security. It is not obvious how to enable powerful discovery query support and collective collaborative functionality that operate on the distributed system as a whole, rather than on a given part of it. Further, it is not obvious how to allow for search results that are fresh, allowing time-sensitive dynamic content. It appears that a Peer-to-Peer (P2P) database network may be well suited to support dynamic distributed database search, for example for service discovery. The key problems then are:

- *What are the detailed architecture and design options for P2P database searching in the context of service discovery? What response models can be used to return matching query results? How should a P2P query processor be organized? What query types can be answered (efficiently) by a P2P network? What query types have the potential to immediately start piping in (early) results? How can a maximum of results be delivered reliably within the time frame desired by a user, even if a query type does not support pipelining? How can loops be detected reliably using timeouts? How can a query scope be used to exploit topology characteristics in answering a query?*
- *Can we devise a unified P2P database framework for general-purpose query support in large heterogeneous distributed systems spanning many administrative domains? More precisely, can we devise a framework that is unified in the sense that it allows to express specific discovery applications for a wide range of data types, node topologies, query languages, query response modes, neighbor selection policies, pipelining characteristics, timeout and other scope options?*

In this section we take the first steps towards unifying the fields of database management systems and P2P computing, which so far have received considerable, but separate, attention. We extend database concepts and practice to cover P2P search. Similarly, we extend P2P concepts and practice to support powerful general-purpose query languages such as XQuery [23] and SQL [33]. As a result, we propose the *Unified Peer-to-Peer Database Framework (UPDF)* and corresponding *Peer Database Protocol (PDP)* for general-purpose query support in large heterogeneous distributed systems spanning many administrative domains. They are unified in the sense that they allow to express specific discovery applications for a wide range of data types, node topologies, query languages, query response modes, neighbor selection policies, pipelining characteristics, timeout and other scope options.

### 4.1. Routed vs. Direct Response, Metadata Responses

When any *originator* wishes to search a P2P network with some query, it sends the query to an *agent node*. The node applies the query to its local database and returns matching results; it also forwards the query to select *neighbor nodes*. These neighbors return their local query results; they also forward the query to select neighbors, and so on. We propose to distinguish four techniques to return matching query results to an originator: *Routed Response*, *Direct*

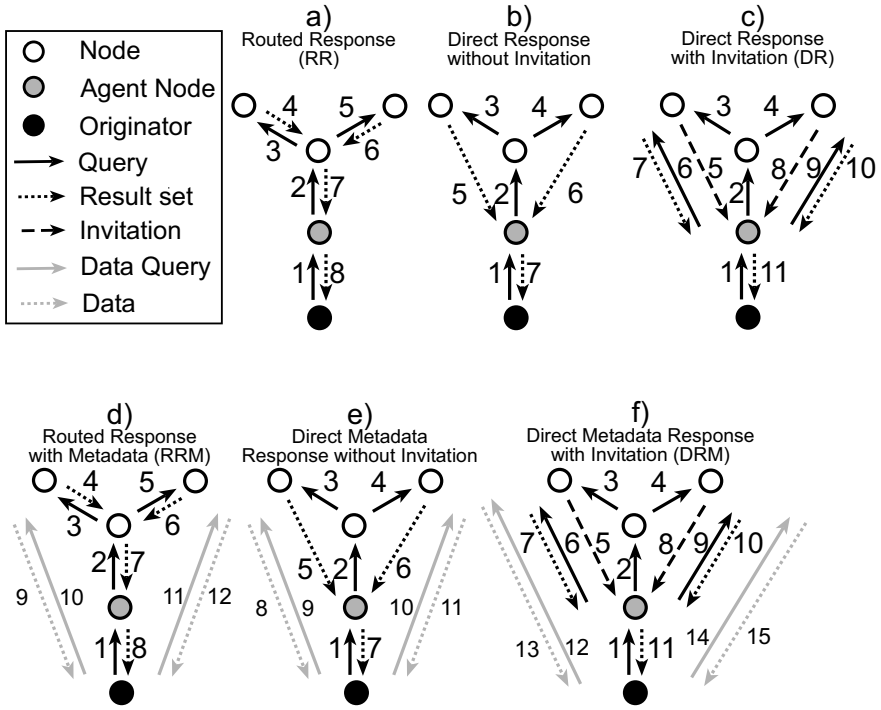



Figure 9. Peer-to-Peer Response Modes.

*Response*, *Routed Metadata Response*, and *Direct Metadata Response*, as depicted in Figure 9. Let us examine the main implications with a Gnutella use case. A typical Gnutella query such as *"Like a virgin"* is matched by some hundreds of files, most of them referring to replicas of the very same music file. Not all matching files are identical because there exist multiple related songs (e.g. remixes, live recordings) and multiple versions of a song (e.g. with different sampling rates). A music file has a size of at least several megabytes. Many thousands of concurrent users submit queries to the Gnutella network.

- **Routed Response.** (Figure 9-a). Results are propagated back into the originator along the paths on which the query flowed outwards. Each (passive) node returns to its (active) client not only its own local results but also all remote results it receives from neighbors. Routing messages through a logical overlay network of P2P nodes is much less efficient than routing through a physical network of IP routers [34]. Routing back even a single Gnutella file (let alone all results) for each query through multiple nodes would consume large amounts of overall system bandwidth, most likely grinding Gnutella to a screeching halt. As the P2P network grows, it is fragmented because nodes with low bandwidth

connections cannot keep up with traffic [35]. Consequently, routed responses are not well suited for file sharing systems such as Gnutella. In general, *overall economics* dictate that routed responses are not well suited for systems that return many and/or large results.

- **Direct Response With and Without Invitation.** To better understand the underlying idea, we first introduce the simpler variant, which is Direct Response Without Invitation (Figure 9-b). Results are not returned by routing back through intermediary nodes. Each (active) node that has local results sends them directly to the (passive) agent, which combines and hands them back to the originator. Response traffic does not travel through the P2P system. It is offloaded via individual point-to-point data transfers on the edges of the network. Let us examine the main implications with a use case.
  As already mentioned, a typical Gnutella query such as *"Like a virgin"* is matched by some hundreds of files, most of them referring to replicas of the very same music file. For Gnutella users it would be sufficient to receive just a small subset of matching files. Sending back *all* such files would unnecessarily consume large amounts of direct bandwidth, most likely restricting Gnutella to users with excessive cheap bandwidth at their disposal. Note however, that the overall Gnutella system would be only marginally affected by a single user downloading, say, a million music files, because the largest fraction of traffic does not travel through the P2P system itself.
  In general, *individual economics* dictate that direct responses without invitation are not well suited for systems that return many equal and/or large results, while a small subset would be sufficient. A variant based on invitation (Figure 9-c) softens the problem by inverting control flow. Nodes with matching files do not blindly push files to the agent. Instead, they invite the agent to initiate downloads. The agent can then act as it sees fit. For example, it can filter and select a subset of data sources and files and reject the rest of the invitations. Due to its inferiority, the variant without invitation is not considered any further. In the remainder of this chapter, we use the term Direct Response as a synonym for Direct Response With Invitation.

- **Routed Metadata Response and Direct Metadata Response.** Here, interaction consists of two phases. In the first phase, routed responses (Figure 9-d) or direct responses (Figure 9-e,f)) are used. However, nodes do not return data results in response to queries, but only small metadata results. The metadata contains just enough information to enable the originator to retrieve the data results and possibly to apply filters before retrieval. In the second phase, the originator selects, based on the metadata, which data results are relevant. The (active) originator directly connects to the relevant (passive) data sources and asks for data results. Again, the largest fraction of response traffic does not travel through the P2P system. It is offloaded via individual point-to-point data transfers on the edges of the network.
  The routed metadata response approach is used by file sharing systems such as Gnutella. A Gnutella query does not return files; it just returns an annotated set of HTTP URLs. The originator connects to a subset of these URLs to download files as it sees fit. Another example is a service discovery system where the first phase returns a set of service links instead of full service descriptions. In the second phase, the originator connects to a subset of these service links to download service descriptions as it sees fit. Another

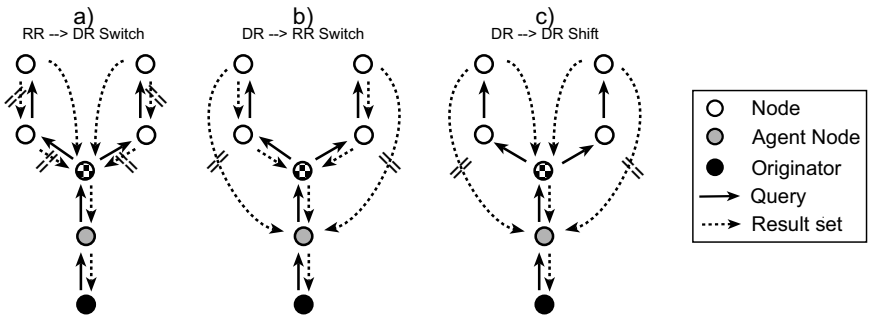



Figure 10. Response Mode Switches and Shifts (RR...Routed Response, DR...Direct Response).

example is a *referral* system where the first phase uses routed metadata response to return the service links of the set of nodes having local matching results ("*Go ask these nodes for the answer*"). In the second phase, the originator or agent connects directly to a subset of these nodes to query and retrieve result sets as it sees fit. This variant avoids the "invitation storm" possible under Direct Response. Referrals are also known as *redirections*. A metadata response mode with a radius scope of zero can be used to implement the referral behavior of the Domain Name System (DNS).

For a detailed comparison of the properties of the various response models, see our prior studies [36]. Although from the functional perspective all response modes are equivalent, no mode is optimal under all circumstances. The question arises as to what extent a given P2P network must mandate the use of any particular response mode throughout the system. Observe that nodes are autonomous and defined by their interface only. Consequently, we propose that response modes can be mixed by *switches* and *shifts*, in arbitrary permutations, as depicted in Figure 10. The response flows that would have been taken are shown crossed out. It is useful to allow specifying as part of the query a hint that indicates the preferred response mode (`routed` or `direct`).

### 4.2. Query Processing

In a distributed database system, there exists a single local database and zero or more neighbors. A classic centralized database system is a special case where there exists a single local database and zero neighbors. From the perspective of query processing, a P2P database system has the same properties as a distributed database system, in a recursive structure. Hence, we propose to organize the P2P query engine like a general distributed query engine [37, 12]. A given query involves a number of operators (e.g. SELECT, UNION, CONCAT, SORT, JOIN, SEND, RECEIVE, SUM, MAX, IDENTITY) that may or may not be exposed at the query language level. For example, the SELECT operator takes a set and returns a new set with tuples satisfying a given predicate. The UNION operator computes the union

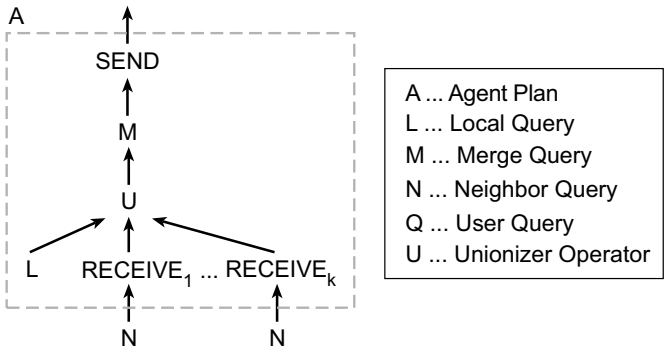



Figure 11. Template Execution Plan.

of two or more sets. The CONCAT operator concatenates the elements of two or more sets into a multiset of arbitrary order (without eliminating duplicates). The IDENTITY operator returns its input set unchanged. The semantics of an operator can be satisfied by several operator implementations, using a variety of algorithms, each with distinct resource consumption, latency and performance characteristics. The query optimizer chooses an efficient query execution plan, which is a tree plugged together from operators. In an execution plan, a parent operator consumes results from child operators.

***Template Query Execution Plan.*** *Any* query `Q` within our query model can be answered by an agent with the *template execution plan* `A` depicted in Figure 11. The plan applies a local query `L` against the tuple set of the local database. Each neighbor (if any) is asked to return a result set for (the same) neighbor query `N`. Local and neighbor result sets are unionized into a single result set by a unionizer operator `U` that must take the form of either `UNION` or `CONCAT`. A merge query `M` is applied that takes as input the result set and returns a new result set. The final result set is sent to the client, i.e. another node or an originator.

***Centralized Execution Plan.*** To see that indeed any query against any kind of database system can be answered within this framework, we derive a simple *centralized execution plan* that always satisfies the semantics of any query `Q`. The plan substitutes specific subplans into the template plan `A`, leading to distinct plans for the agent node (Figure 12-a) and neighbors nodes (Figure 12-b). In the case of XQuery and SQL, parameters are substituted as follows:

| **XQuery** | **SQL** |
|---|---|
| `A: M=Q, U=UNION, L="RETURN /", N'=N`<br>`N: M=IDENTITY, U=UNION, L="RETURN /", N'=N` | `A: M=Q, U=UNION, L="SELECT *", N'=N`<br>`N: M=IDENTITY, U=UNION, L="SELECT *", N'=N` |

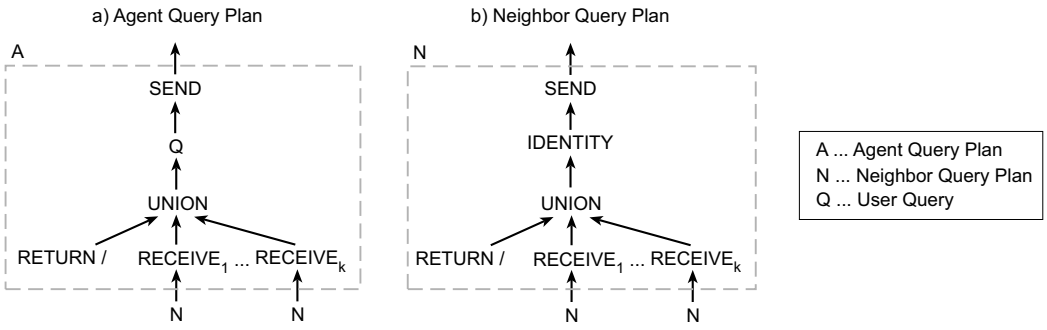



Figure 12. Centralized Execution Plan.

In other words, the agent's plan `A` fetches all raw tuples from the local and all remote databases, unionizes the result sets, and then applies the query `Q`. Neighbors are handed a rewritten neighbor query `N` that recursively fetches all raw tuples, and returns their union. The neighbor query `N` is recursively partitionable (see below).

The same centralized plan works for routed and direct response, both with and without metadata. Under direct response, a node does forward the query `N`, but does not attempt to receive remote result sets (conceptually empty result sets are delivered). The node does not send a result set to its predecessor, but directly back to the agent.

The centralized execution plan can be inefficient because potentially large amounts of base data have to be shipped to the agent before locally applying the user's query. However, sometimes this is the only plan that satisfies the semantics of a query. This is always the case for a complex query. A more efficient execution plan can sometimes be derived (as proposed below). This is always the case for a simple and medium query.

***Recursively Partitionable Query.*** A P2P network can be efficient in answering queries that are recursively partitionable. A query `Q` is *recursively partitionable* if, for the template plan `A`, there exists a merge query `M` and a unionizer operator `U` to satisfy the semantics of the query `Q` assuming that `L` and `N` are chosen as `L = Q` and `N = A`. In other words, a query is recursively partitionable if the very same execution plan *can* be recursively applied at every node in the P2P topology. The corresponding execution plan is depicted in Figure 13.

The input and output of a merge query have the same form as the output of the local query `L`. Query processing can be parallelized and spread over all participating nodes. Potentially very large amounts of information can be searched while investing little resources such as processing time per individual node. The recursive parallel spread of load implied by a recursively partitionable query is the basis of the massive P2P scalability potential. However, query performance is not necessarily good, for example due to high network I/O costs.

Now we are in the position to clarify the definition of simple, medium and complex queries.

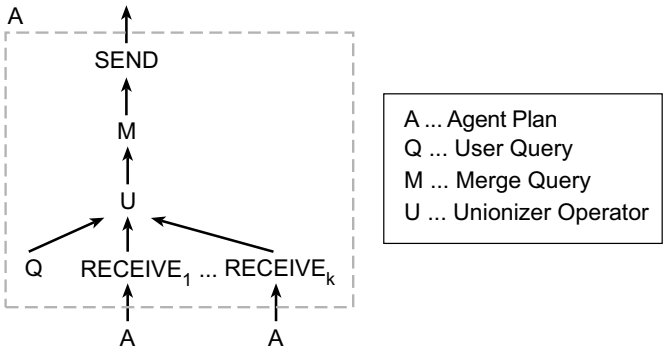



Figure 13. Execution Plan for Recursively Partitionable Query.

- *Simple Query.* A query is *simple* if it is recursively partitionable using `M = IDENTITY, U = UNION`. An example is *Find all (available) services.*
- *Medium Query.* A query is a *medium* query if it is not simple, but it is recursively partitionable. An example is *Return the number of replica catalog services.*
- *Complex Query.* A query is *complex* if it is not recursively partitionable. An example is *Find all (execution service, storage service) pairs where both services of a pair live within the same domain.*

For simplicity, in the remainder of this chapter we assume that the user explicitly provides `M` and `U` along with a query `Q`. If `M` and `U` are not provided as part of a query to any given node, the node acts defensively by assuming that the query is not recursively partitionable. Choosing `M` and `U` is straightforward for a human being. Consider for example the following medium XQueries.

- *Return the number of replica catalog services.* The merge query computes the sum of a set of numbers. The unionizer is CONCAT.

```
Q = RETURN
      <tuple>
        count(/tupleset/tuple/content/service[interface/@type="repcat"])
      </tuple>
M = RETURN
      <tuple>
        sum(/tupleset/tuple)
      </tuple>
U = CONCAT
```

- *Find the service with the largest uptime.*

```
Q=M= RETURN (/tupleset/tuple[@type="service"] SORTBY (./@uptime)) [last()]
U =  UNION
```

Note that the query engine always encapsulates the query output with a `tupleset` root element. A query need not generate this root element as it is implicitly added by the environment.

***Pipelining.*** The success of many applications depends on how fast they can start producing initial/relevant portions of the result set rather than how fast the entire result set is produced [38]. Often an originator would be happy to already work with one or a few *early results*, as long as they arrive quickly and reliably. Results that arrive later can be handled later, or are ignored anyway. This is particularly often the case in distributed systems where many nodes are involved in query processing, each of which may be unresponsive for many reasons. The situation is even more pronounced in systems with loosely coupled autonomous nodes.

Operators of any kind have a uniform iterator interface, namely the three methods `open()`, `next()` and `close()`. For efficiency, the method `next()` can be asked to deliver several results at once in a so-called *batch*. Semantics are as follows: *"Give me a batch of at least* `N` *and at most* `M` *results"* (less than `N` results are delivered when the entire query result set is exhausted). For example, the SEND and RECEIVE network communication operators typically work in batches.

The monotonic semantics of certain operators such as SELECT, UNION, CONCAT, SEND, RECEIVE allow that operator implementations consume just one or a few child results on `next()`. In contrast, the non-monotonic semantics of operators such as SORT, GROUP, MAX, some JOIN methods, etc. require that operator implementations consume *all* child results already on `open()` in order to be able to deliver a result on the first call to `next()`. Since the output of these operators on a subset of the input is not, in general, a subset of the output on the whole input, these operators need to see all of their input before they produce the correct output. This does not break the iterator concept but has important latency and performance implications. Whether the root operator of an agent exhibits a short or long latency to deliver to the originator the first result from the result set depends on the query operators in use, which in turn depend on the given query. In other words, for some query types the originator has the potential to immediately start piping in results (at moderate performance rate), while for other query types it must wait for a long time until the first result becomes available (the full result set arrives almost at once, however).

A query (an operator implementation) is said to be *pipelined* if it can already produce at least one result tuple before all input tuples have been seen. Otherwise, a query (an operator) is said to be *non-pipelined*. Simple queries do support pipelining (e.g. Gnutella queries). Medium queries may or may not support pipelining, whereas complex queries typically do not support pipelining. Figure 14 depicts pipelined and non-pipelined example queries.

### 4.3. Static Loop Timeout and Dynamic Abort Timeout

Clearly, there comes a time when a user is no longer interested in query results, no matter whether any more results might be available. The query roaming the network and its response traffic should fade away after some time. In addition, P2P systems are well advised to attempt to limit resource consumption by defending against *runaway* queries roaming forever or producing gigantic result sets, either unintended or malicious. To address these problems,

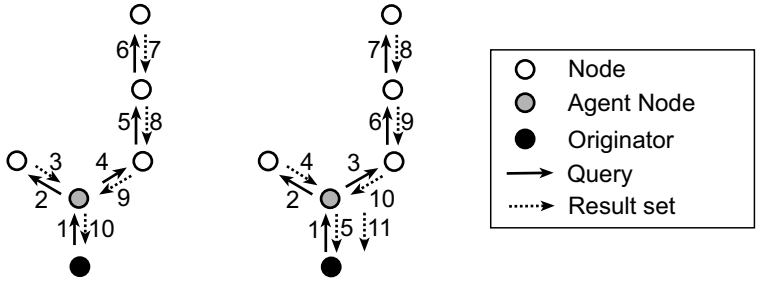



Figure 14. Non-Pipelined (left) and Pipelined Query (right).

an absolute *abort timeout* is attached to a query, as it travels across hops. An abort timeout can be seen as a deadline. Together with the query, a node tells a neighbor *"I will ignore (the rest of) your result set if I have not received it before 12:00:00 today."* The problem, then, is to ensure that a maximum of results can be delivered reliably within the time frame desired by a user. The value of a *static timeout* remains unchanged across hops, except for defensive modification in flight triggered by runaway query detection (e.g. infinite timeout). In contrast, it is intended that the value of a *dynamic timeout* be decreased at each hop. Nodes further away from the originator may time out earlier than nodes closer to the originator.

***Dynamic Abort Timeout.*** A static abort timeout is entirely unsuitable for non-pipelined result set delivery, because it leads to a serious reliability problem, which we propose to call *simultaneous abort timeout.* If just one of the many nodes in the query path fails to be responsive for whatever reasons, all other nodes in the path are waiting, eventually time out and attempt to return at least a partial result set. However, it is impossible that any of these partial results ever reach the originator, because all nodes time out *simultaneously* (and it takes some time for results to flow back).

To address the simultaneous abort timeout problem, we propose dynamic abort timeouts. Under *dynamic abort timeout*, nodes further away from the originator time out earlier than nodes closer to the originator. This provides some safety time window for the partial results of any node to flow back across multiple hops to the originator. Intermediate nodes can and should adaptively decrease the timeout value as necessary, in order to leave a large enough time window for receiving and returning partial results subsequent to timeout.

Observe that the closer a node is to the originator, the more important it is (if it cannot meet its deadline, results from a large branch are discarded). Further, the closer a node is to the originator, the larger is its response and bandwidth consumption. Thus, as a good policy to choose the safety time window, we propose *exponential decay with halving.* The window size is halved at each hop, leaving large safety windows for important nodes and tiny window sizes for nodes that contribute only marginal result sets. Also, taking into account network latency and the time it takes for a query to be locally processed, the timeout is updated at each hop `N` according to the following recurrence formula:

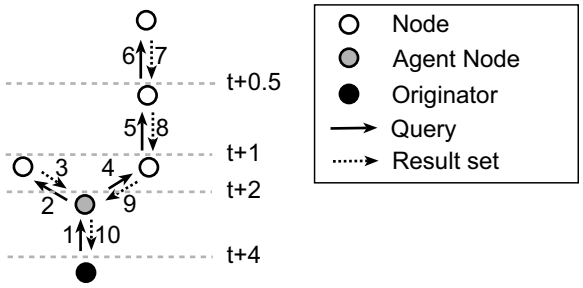



Figure 15. Dynamic Abort Timeout.

$$timeout_N = currenttime_N + \frac{timeout_{N-1} - currenttime_N}{2} \quad (1)$$

Consider for example Figure 15. At time `t` the originator submits a query with a dynamic abort timeout of `t+4` seconds. In other words, it warns the agent to ignore results after time `t+4`. The agent in turn intends to safely meet the deadline and so figures that it needs to retain a safety window of 2 seconds, already starting to return its (partial) results at time `t+2`. The agent warns its own neighbors to ignore results after time `t+2`. The neighbors also intend to safely meet the deadline. From the 2 seconds available, they choose to allocate 1 second, and leave the rest to the branch remaining above. Eventually, the safety window becomes so small that a node can no longer meet a deadline on timeout. The results from the unlucky node are ignored, and its partial results are discarded. However, other nodes below and in other branches are unaffected. Their results survive and have enough time to hop all the way back to the originator before time `t+4`.

***Static Loop Timeout.*** The same query may arrive at a node multiple times, along distinct routes, perhaps in a complex pattern. For reliable loop detection, a query has an identifier and a certain life time. To each query, an originator attaches a *loop timeout* and a different *transaction identifier*, which is a universally unique identifier (UUID). A node maintains a state table of transaction identifiers and returns an error when a query is received that has already been seen and has not yet timed out. On loop timeout, a node may "forget" about a query by deleting it from the state table. To be able to reliably detect a loop, a node must not forget a transaction identifier before its loop timeout has been reached. Interestingly, a static loop timeout is required in order to fully preserve query semantics. Otherwise, a problem arises that we propose to call *non-simultaneous loop timeout.* The non-simultaneous loop timeout problem is caused by the fact that some nodes still forward the query to other nodes when the destinations have already forgotten it. In other words, the problem is that loop timeout does not occur simultaneously everywhere. Consequently, a loop timeout must be static (does not change across hops) to guarantee that loops can reliably be detected. Along with a query, an originator not only provides a dynamic abort timeout, but also a static loop timeout. Initially

at the originator, both values must be identical (e.g. `t+4`). After the first hop, both values become unrelated.

To summarize, we have `abort timeout` $\leq$ `loop timeout`. To ensure reliable loop detection, a loop timeout must be static whereas an abort timeout may be static or dynamic. Under non-pipelined result set delivery, dynamic abort timeout using *exponential decay with halving* ensure that a maximum of results can be delivered reliably within the time frame desired by a user. We speculate that dynamic timeouts could also incorporate sophisticated cost functions involving latency and bandwidth estimation and/or economic models.

### 4.4. Query Scope

As in a data integration system, the goal is to exploit several independent information sources as if they were a single source. This is important for distributed systems in which node topology or deployment model change frequently. For example, cross-organizational Grids and P2P networks exhibit such a character. However, in practice, it is often sufficient (and much more efficient) for a query to consider only a subset of all tuples (service descriptions) from a subset of nodes. For example, a typical query may only want to search tuples (services) within the scope of the domain `cern.ch` and ignore the rest of the world. To this end, we cleanly separate the concepts of (logical) *query* and (physical) *query scope*. A query is formulated against a global database view and is insensitive to link topology and deployment model. In other words, to a query the set of tuples appears as a single homogenous database, even though the set may be (recursively) partitioned across many nodes and databases. This means that in a relational or XML environment, at the global level, the set of all tuples *appears* as a single, very large, table or XML document, respectively. The query scope, on the other hand, is used to navigate and prune the link topology and filter on attributes of the deployment model. Conceptually, the scope is the input fed to the query. The query scope is a set and may contain anything from all tuples in the universe to none. Both query and scope can prune the search space, but they do so in a very different manner. A query scope is specified either *directly* or *indirectly*. One can distinguish scopes based on neighbor selection, timeout and radius.

***Neighbor Selection.*** For simplicity, all our discussions so far have implicitly assumed a *broadcast* model (on top of TCP) in which a node forwards a query to all neighbor nodes. However, in general one can select a subset of neighbors, and forward concurrently or sequentially. Fewer query forwards lead to less overall resource consumption. The issue is critical due to the snowballing (epidemic, flooding) effect implied by broadcasting. Overall bandwidth consumption grows exponentially with the query radius, producing enormous stress on the network and drastically limiting its scalability [39, 34].

Clearly selecting a neighbor subset can lead to incomplete coverage, missing important results. The best policy to adopt depends on the context of the query and the topology. For example, the scope can select only neighbors with a service description of interface type "Gnutella". In an attempt to explicitly exploit topology characteristics, a virtual organization of a Grid may deliberately organize global, intermediate and local job schedulers into a tree-like topology. Correct operation of scheduling may require reliable discovery of all or at least most relevant schedulers in the tree. In such a scenario, random selection of half of the neighbors

at each node is certainly undesirable. A policy that selects all `child` nodes and ignores all `parent` nodes may be more adequate. Further, a node may maintain statistics about its neighbors. One may only select neighbors that meet minimum requirements in terms of latency, bandwidth or historic query outcomes (`maxLatency, minBandwidth, minHistoricResult`). Other node properties such as hostname, domain name, owner, etc. can be exploited in query scope guidance, for example to implement security policies. Consider an example where the scheduling system may only trust nodes from a select number of security domains. Here a query should never be forwarded to nodes not matching the trust pattern.

Further, in some systems, finding a single result is sufficient. In general, a user or any given node can guard against unnecessarily large result sets, message sizes and resource consumption by specifying the maximum number of result tuples (`maxResults`) and bytes (`maxResultsBytes`) to be returned. Using sequential propagation, depending on the number of results already obtained from the local database and a subset of the selected neighbors, the query may no longer need to be forwarded to the rest of the selected neighbors.

***Neighbor Selection Query.*** For flexibility and expressiveness, we propose to allow the user to specify the selection policy. In addition to the normal query, the user defines a *neighbor selection query* (XQuery) that takes the tuple set of the current node as input and returns a subset that indicates the nodes selected for forwarding. For example, a neighbor query implementing broadcasting selects all services with registry and P2P query capabilities, as follows:

```
RETURN /tupleset/tuple[@type="service"
  AND content/service/interface[@type="Consumer-1.0"]
  AND content/service/interface[@type="XQuery-1.0"]]
```

A wide range of policies can be implemented in this manner. The neighbor selection policy can draw from the rich set of information contained in the tuples published to the node. Further, recall that the set of tuples in a database may not only contain service descriptions of neighbor nodes (e.g. in WSDL [4] or SWSDL [6]), but also other kind of (soft state) content published from any kind of content provider. For example, this may include the type of queries neighbor nodes can answer, descriptions of the kind of tuples they hold (e.g. their types), or a compact summary or index of their content. Content available to the neighbor selection query may also include host and network information as well as statistics that a node periodically publishes to its immediate neighbors. A neighbor selection query enables group communication to all nodes with certain characteristics (e.g. the same group ID). For example, broadcast and random selection can be expressed with a neighbor query. One can select nodes that support given interfaces (e.g. Gnutella [13], Freenet [14] or job scheduling). In a tree topology, a policy can use the tuple `context` attribute to select all `child` nodes and to ignore all `parent` nodes. One can implement domain filters and security filters (e.g. `allow/deny` regular expressions as used in the Apache HTTP server if the tuple set includes metadata such as hostname and node owner. Power-law policies [40] can be expressed if metadata includes the number of neighbors to the `n`-th radius. To summarize, a neighbor selection query can be used to implement *smart dynamic routing*.

***Radius.*** The *radius* of a query is a measure of path length. More precisely, it is the maximum number of hops a query is allowed to travel on any given path. The radius is decreased by one at each hop. The roaming query and response traffic must fade away upon reaching a radius of less than zero. A scope based on radius serves similar purposes as a timeout. Nevertheless, timeout and radius are complementary scope features. The radius can be used to indirectly limit result set size. In addition, it helps to limit latency and bandwidth consumption and to guard against runaway queries with infinite lifetime. In Gnutella and Freenet, the radius is the primary means to specify a query scope. The radius is termed *TTL (time-to-live)* in these systems. Neither of these systems support timeouts.

For maximum result set size limiting, a timeout and/or radius can be used in conjunction with neighbor selection, routed response, and perhaps sequential forward, to implement the *expanding ring* [41] strategy. The term stems from IP multicasting. Here an agent first forwards the query to a small radius/timeout. Unless enough results are found, the agent forwards the query again with increasingly large radius/timeout values to reach further into the network, at the expense of increasingly large overall resource consumption. On each expansion radius/timeout are multiplied by some factor.

## 5. Peer Database Protocol

In this section we summarize how the operations of the Unified Peer-to-Peer Database Framework (UPDF) and `XQuery` interface from Section 3.1 are carried over (bound to) our *Peer Database Protocol (PDP)* [6, 42]. PDP supports centralized and P2P database queries for a wide range of database architectures and response models such that the stringent demands of ubiquitous Internet infrastructures in terms of scalability, efficiency, interoperability, extensibility and reliability can be met. Any client (e.g. an originator or a node) can use PDP to query the P2P network, and to retrieve the corresponding result set in an iterator-style. While the use of PDP for communication between nodes is mandatory to achieve interoperability, any arbitrary additional protocol and interface may be used for communication between an originator and a node (e.g. a simple stateless SOAP/HTTP request-response or shared memory protocol). For flexibility and simplicity, and to allow for gatewaying, mediation and protocol translation, the relationship between an originator and a node may take any arbitrary form, and is therefore left unspecified.

The high-level messaging model employs four request messages (QUERY, RECEIVE, INVITE, CLOSE) and a response message (SEND). A *transaction* is a sequence of one or more message exchanges between two peers (nodes) for a given query. An example transaction is a QUERY-RECEIVE-SEND-RECEIVE-SEND-CLOSE sequence. A peer can concurrently handle multiple independent transactions. A transaction is identified by a transaction identifier. Every message of a given transaction carries the same transaction identifier.

A QUERY message is asynchronously forwarded along hops through the topology. A RECEIVE message is used by a client to request query results from another node. It requests the node to respond with a SEND message, containing a batch of at least $N$ and at most $M$ results from the (remainder of the) result set. A client may issue a CLOSE request to inform a node that the remaining results (if any) are no longer needed and can safely be discarded. Like a

QUERY, a CLOSE is asynchronously forwarded. If the local result set is not empty under direct response, the node directly contacts the agent with an INVITE message to solicit a RECEIVE message. A RECEIVE request can ask to deliver SEND messages in either synchronous (pull) or asynchronous (push) mode. In synchronous mode a single RECEIVE request must precede every single SEND response. An example sequence is RECEIVE-SEND-RECEIVE-SEND. In asynchronous mode a single RECEIVE request asks for a sequence of successive SEND responses. A client need not explicitly request more results, as they are automatically pushed in a sequence of zero or more SENDs. An example sequence is RECEIVE-SEND-SEND-SEND. Appropriately sized batched delivery greatly reduces the number of hops incurred by a single RECEIVE. To reduce latency, a node may prefetch query results.

Discrete messages belong to well-defined message exchange patterns. For example, the pattern of synchronous exchanges (one-to-one, pull) is supported as well as the pattern of asynchronous exchanges (one-to-many, push). For example, the response to a MSG RECEIVE may be an *error* (ERR), a *reply* (RPY SEND) or a sequence of zero or more *answers* (ANS SEND), followed by a *null terminator* message (NULL). The RPY OK and ERR message type are introduced because any realistic messaging model must deal with acknowledgments and errors. The following message exchanges are permitted:

```
MSG_QUERY   --> RPY_OK | ERR
MSG_RECEIVE --> RPY_SEND | (ANS_SEND [0:N], NULL) | ERR
MSG_INVITE  --> RPY_OK | ERR
MSG_CLOSE   --> RPY_OK | ERR
```

For simplicity and flexibility, PDP uses straightforward XML representations for messages, as depicted in Figure 16. Without loss of generality, example query expressions (e.g. user query, merge query and neighbor selection query) are given in the XQuery language [23]. Other query languages such as XPath, SQL, LDAP [43] or subscription interest statements could also be used. Indeed, the messages and network interactions required to support efficient P2P publish-subscribe and event trigger systems do not differ at all from the ones presented above.

PDP has a number of key properties. It is applicable to any node topology (e.g. centralized, distributed or P2P) and to multiple P2P response modes (routed response and direct response, both with and without metadata modes). To support loosely coupled autonomous Internet infrastructures, the model is connection-oriented (ordered, reliable, congestion sensitive) and message-oriented (loosely coupled, operating on structured data). For efficiency, it is stateful at the protocol level, with a transaction consisting of one or more discrete message exchanges related to the same query. It allows for low latency, pipelining, early and/or partial result set retrieval due to synchronous pull, and result set delivery in one or more variable sized batches. It is efficient, due to asynchronous push with delivery of multiple results per batch. It provides for resource consumption and flow control on a per query basis, due to the use of a distinct channel per transaction. It is scalable, due to application multiplexing, which allows for very high query concurrency and very low latency, even in the presence of secure TCP connections. To encourage interoperability and extensibility it is fully based on the BEEP [31] Internet Engineering Task Force (IETF) standard, for example in terms of asynchrony, encoding, framing, authentication, privacy and reporting. Finally, we note that SOAP can be carried over BEEP in a straightforward manner [32], and that BEEP, in turn, can be carried over any reliable transport layer (TCP is merely the default).

```
<MSG_QUERY transactionID = "12345">
  <query>
        <userquery> RETURN /tupleset/tuple </userquery>
        <mergequery unionizer="UNION"> RETURN /tupleset/tuple </mergequery>
  </query>
  <scope loopTimeout = "2000000000000" abortTimeout = "1000000000000"
         logicalRadius = "7" physicalRadius = "4"
         maxResults = "100"  maxResultsBytes = "100000">
         <neighborSelectionQuery>         <!-- implements broadcasting -->
            RETURN /tupleset/tuple[@type="service"
              AND content/service/interface[@type="Consumer-1.0"]
              AND content/service/interface[@type="XQuery-1.0"]]
         </neighborSelectionQuery>
  </scope>
  <options>
         <responseMode> routed </responseMode>
         <originator> fred@example.com </originator>
  </options>
</MSG_QUERY>

<MSG_RECEIVE transactionID = "12345">
  <mode minResults = "1" maxResults = "10">  synchronous  </mode>
</MSG_RECEIVE>

<RPY_SEND transactionID = "12345">
  <data nonBlockingResultsAvailable = "-1" estimatedResultsAvailable = "-1">
    <tupleset TS4="100">
       <tuple link="http://sched.infn.it:8080/pub/getServiceDescription"
              type="service" ctx="child" TS1="20" TC="25" TS2="30" TS3="40">
          <content>
             <service> service description B goes here </service>
          </content>
       </tuple>
       ... more tuples can go here ...
    </tupleset>
  </data>
</RPY_SEND>

<ANS_SEND transactionID = "12345">
    structure is identical to RPY_SEND (see above) ...
</ANS_SEND>

<MSG_INVITE transactionID = "12345">
  <avail nonBlockingResultsAvailable="50" estimatedResultsAvailable="100"/>
</MSG_INVITE>

<MSG_CLOSE transactionID = "12345" code="555"> maximum idle time exceeded </MSG_CLOSE>

<RPY_OK transactionID = "12345"/>
<ERR transactionID = "12345" code="550"> transaction identifier unknown </ERR>
```

Figure 16. Example Messages of Peer Database Protocol.

## 6. Related Work

***RDBMS.*** (Distributed) Relational database systems [12] provide SQL as a powerful query language. They assume tight and consistent central control and hence are infeasable in Grid environments, which are characterized by heterogeneity, scale, lack of central control, multiple autonomous administrative domains, unreliable components and frequent dynamic change. They do not support an XML data model and the XQuery language. Further, they do not provide dynamic content generation, soft state based publication and content caching. RDBMS are not designed for use in P2P systems. For example, they do not support asynchronous push, invitations, scoping, neighbor selection and dynamic timeouts. Our work does not compete with an RDBMS, though. A node may well internally use an RDBMS for data management. A node can accept queries over an XML view and internally translate the query into SQL [44, 45]. An early attempt towards a WAN distributed DBMS was Mariposa [46], designed for scalability to many cooperating sites, data mobility, no global synchronization and local autonomy. It used an economic model and bidding for adaptive query processing, data placement and replication.

***ANSA and CORBA.*** The ANSA project was an early collaborative industry effort to advance distributed computing. It defined trading services [47] for advertizement and discovery of relevant services, based on service type and simple constraints on attribute/value pairs. The CORBA Trading service [48] is an evolution of these efforts.

***UDDI.*** UDDI (Universal Description, Discovery and Integration) [8] is an emerging industry standard that defines a business oriented access mechanism to a centralized registry holding XML based WSDL service descriptions. UDDI is a definition of a specific service class, not a discovery architecture. It does not offer a dynamic data model. It is not based on soft state, which limits its ability to dynamically manage and remove service descriptions from a large number of autonomous third parties in a reliable, predictable and simple way. Query support is rudimentary. Only key lookups with primitive qualifiers are supported, which is insufficient for realistic service discovery use cases.

***Jini, SLP, SDS, INS.*** The centralized Jini Lookup Service [49] is located by Java clients via a UDP multicast. The network protocol is not language independent because it relies on the Java-specific object serialization mechanism. Publication is based on soft state. Clients and services must renew their leases periodically. Content freshness is not addressed. The query "language" allows for simple string matching on attributes, and is even less powerful than LDAP.

The Service Location Protocol (SLP) [50] uses multicast, softstate and simple filter expressions to advertize and query the location, type and attributes of services. The query "language" is more simple than Jini's. An extension is the Mesh Enhanced Service Location Protocol (mSLP) [51], increasing scalability through multiple cooperating directory agents. Both assume a single administrative domain and hence do not scale to the Internet and Grids.

The Service Discovery Service (SDS) [52] is also based on multi cast and soft state. It supports a simple XML based exact match query type. SDS is interesting in that it mandates secure channels with authentication and traffic encryption, and privacy and authenticity of

service descriptions. SDS servers can be organized in a distributed hierarchy. For efficiency, each SDS node in a hierarchy can hold an index of the content of its sub-tree. The index is a compact aggregation and custom tailored to the narrow type of query SDS can answer. Another effort is the Intentional Naming System [53]. Like SDS, it integrates name resolution and routing.

***JXTA.*** The goal of the JXTA P2P network [54, 55, 56] is to have peers that can cooperate to form self-organized and self-configured peer groups independent of their position in the network, and without the need of a centralized management infrastructure. JXTA defines six stateless best-effort protocols for ad hoc, pervasive, and multi-hop P2P computing. These are designed to run over uni-directional, unreliable transports. Due to this ambitious goal, a range of well-known higher level abstractions (e.g. bi-directional secure messaging) are (re)invented from first principles.

The Endpoint Routing Protocol allows to discover a route (sequence of hops) from one peer to another peer, given the destination peer ID. The Rendezvous Protocol offers publish-subscribe functionality within a peer group. The Peer Resolver Protocol and Peer Discovery Protocol allow for publication of advertizements and *simple* queries that are unreliable, stateless, non-pipelined, and non-transactional. We believe that this limits scalability, efficiency and applicability for service discovery and other non-trivial use cases. Lacking expressive means for query scoping, neighbor selection and timeouts, it is unclear how chained rendezvous peers can form a search network. We believe that JXTA Peer Groups, JXTA search and publish/subscribe can be expressed within our UPDF framework, but not vice versa.

***GMA.*** The Grid Monitoring Architecture (GMA) [57, 58] is intended to enable efficient monitoring of distributed components, for example to allow for fault detection and performance prediction. GMA handles performance data transmitted as time-stamped *events*. It consists of three types of components, namely Directory Service, Producer and Consumer. Producers and Consumers publish their existance in a centralized directory service. Consumers can use the directory service to discover producers of interest, and vice versa. GMA briefly sketches three interactions for transferring data between producers and consumers, namely publish/subscribe, query/response and notification. Both consumers and producers can initiate interactions.

GMA neither defines a query language, nor a data model, nor a network protocol. It does not consider the use of multi-hop P2P networks, and hence does not address loop detection, scoping, timeouts and neighbor selection. Synchronous multi-message exchanges and routed responses are not considered. Event data is always asynchronously pushed from a producer directly to a consumer. GMA like server-initiated interactions could be offered via the INVITE message of the Peer Database Protocol, but currently we do not see enough compelling reasons for doing so. For a comparison of various response modes, see our prior studies [36]. We believe that GMA can be expressed within our framework, but not vice versa.

***OGSA.*** The independently emerging *Open Grid Services Architecture (OGSA)* [59, 60] exhibits striking similarities with WSDA, in spirit and partly also in design. However, although it is based on soft state, OGSA does not offer a dynamic data model allowing for dynamic refresh of content. Hence it requires trust delegation on publication, which is problematic

for security sensitive data such as detailed service descriptions. Further, the data model and publication is not based on sets, resulting in scalability problems in the presence of large numbers of tuples. The absence of set semantics also seriously limits the potential for query optimization. The use of arbitrary MIME content is not foreseen, reducing applicability. The concepts of notification and registration are not unified, as in the WSDA `Consumer` interface. OGSA does define an interesting interface for publish/subscribe functionality, but no corresponding network protocol (e.g. such as the Peer Database Protocol). OGSA mandates that every service in the universe must comply with the `GridService` interface, unnecessarily violating the principle of non-disruptiveness. It is unclear from the material whether OGSA intends in the future to support either or both XQuery, XPath, or none. Finally, OGSA does not consider that the set of information tuples in the universe is partitioned over multiple autonomous nodes. Hence, it does not consider Peer-to-Peer networks, and their (non-trivial) implications, for example wrt. query processing, timeouts, pipelining and network protocols. For a detailed comparison of WSDA and OGSA, see [61].

***DNS.*** Distributed databases with a hierarchical name space such as the Domain Name System (DNS) [62] can efficiently answer *simple* queries of the form *"Find an object by its full name"*. Queries are not forwarded (routed) through the (hierarchical) link topology. Instead, a node returns a *referral* message that redirects an originator to the next closer node. The originator explicitly queries the next node, is referred to yet another closer node, and so on. To support neighbor selection in a hierarchical name space within our UPDF framework, a node can publish to its neighbors not only its service link, but also the name space it manages. The DNS referral behavior can be implemented within UPDF by using a radius scope of zero. The same holds for the LDAP referral behavior (see below).

***LDAP and MDS.*** The Lightweight Directory Access Protocol (LDAP) [43] defines a network protocol in which clients send requests to and receive responses from LDAP servers. LDAP is an extensible network protocol, not a discovery architecture. It does not offer a dynamic data model, is not based on soft state and does not follow an XML data model. The expressive power of the LDAP query language is insufficient for realistic service discovery use cases [6]. Like DNS, it supports referrals in a hierarchical namespace but not query forwarding.

The Metacomputing Directory Service (MDS) [63] inherits all properties of LDAP. As a result, its query language is insufficient for service discovery, and it does not follow an XML data model. MDS does not offer a dynamic data model, limiting cache freshness steering. However, it is based on soft state. MDS is not a web service, because it is not specified by a service description language. It does not offer interfaces and operations that may be bound to multiple network protocols. However, it appears that MDS is being recast to fit into the OGSA architecture. Indeed, the OGSA registry and notification interfaces could be seen as new and abstracted clothings for MDS. Beyond LDAP, MDS offers a simple form of query forwarding that allows for multi-level hierarchies but not for arbitrary topologies. It does not support radius and dynamic abort timeout, pipelined query execution across nodes as well as direct response and metadata responses.

***Query Processing.*** *Simple* queries for lookup by key are assumed in most P2P systems such as DNS [62], Gnutella [13], Freenet [14], Tapestry [15], Chord [16] and Globe [17], leading to highly specialized *content-addressable* networks centered around the theme of distributed hash table lookup. *Simple* queries for exact match (i.e. given a flat set of attribute values find all tuples that carry exactly the same attribute values) are assumed in systems such as SDS [52] and Jini [49]. Our approach is distinguished in that it not only supports all of the above query types, but it also supports queries from the rich and expressive general-purpose query languages XQuery [23] and SQL.

***Pipelining.*** For a survey of adaptive query processing, including pipelining, see the special issue of [64]. [65] develops a general framework for producing partial results for queries involving any non-monotonic operator. The approach inserts update and delete directives into the output stream. The Tukwila [66] and Niagara projects [67] introduce data integration systems with adaptive query processing and XML query operator implementations that efficiently support pipelining. Pipelining of hash joins is discussed in [68, 69, 70]. Pipelining is often also termed *streaming* or *non-blocking* execution.

***Neighbor Selection.*** *Iterative deepening* [71] is a similar technique to *expanding ring* where an optimization is suggested that avoids reevaluating the query at nodes that have already done so in previous iterations. Neighbor selection policies that are based on randomness and/or historical information about the result set size of prior queries are simulated and analyzed in [72]. An efficient neighbor selection policy is applicable to simple queries posed to networks in which the number of links of nodes exhibits a power law distribution (e.g. Freenet and Gnutella) [40]. Here most (but not all) matching results can be reached with few hops by selecting just a very small subset of neighbors (the neighbors that themselves have the most neighbors to the n-th radius). Note, however, that the policy is based on the assumption that not all results must be found and that all query results are equally relevant. These related works discuss in isolation neighbor selection techniques for a particular query type, without the context of a framework for comprehensive query support.

## 7. Conclusions

This chapter distills and generalizes the essential properties of the discovery problem and then develops solutions that apply to a wide range of large distributed Internet systems. It shows how to support expressive general-purpose queries over a view that integrates autonomous dynamic database nodes from a wide range of distributed system topologies. We describe the first steps towards the convergence of Grid Computing, Peer-to-Peer Computing, Distributed Databases and Web Services, each of which introduces core concepts and technologies necessary for *Making the Global Infrastructure a Reality*.

Grids are collaborative distributed Internet systems characterized by large scale, heterogeneity, lack of central control, multiple autonomous administrative domains, unreliable components and frequent dynamic change. We address the problems of maintaining dynamic and timely information populated from a large variety of unreliable, frequently changing,

autonomous and heterogeneous remote data sources by designing a database for XQueries over dynamic distributed content – the so-called *hyper registry*. The registry has a number of key properties. An XML data model allows for structured and semi-structured data, which is important for integration of heterogeneous content. The XQuery language allows for powerful searching, which is critical for non-trivial applications. Database state maintenance is based on soft state, which enables reliable, predictable and simple content integration from a large number of autonomous distributed content providers. Content link, content cache and a hybrid pull/push communication model allow for a wide range of dynamic content freshness policies, which may be driven by all three system components: content provider, registry and client.

We propose and specify an open discovery architecture, the *Web Service Discovery Architecture (WSDA)*. WSDA views the Internet as a large set of services with an extensible set of well-defined interfaces. It has a number of key properties. It promotes an interoperable web service layer on top of Internet software, because it defines appropriate services, interfaces, operations and protocol bindings. It embraces and integrates solid industry standards such as XML, XML Schema, SOAP, WSDL and XQuery. It allows for integration of emerging standards such as the Web Service Inspection Language (WSIL). It is modular because it defines a small set of orthogonal multi-purpose communication primitives (building blocks) for discovery. These primitives cover service identification, service description retrieval, data publication as well as minimal and powerful query support. Each communication primitive is deliberately designed to avoid any unnecessary complexity. WSDA is open and flexible because each primitive can be used, implemented, customized and extended in many ways. It is powerful because the individual primitives can be combined and plugged together by specific clients and services to yield a wide range of behaviors and emerging synergies. It is unified because it subsumes an array of disparate concepts, interfaces and protocols under a single semi-transparent umbrella.

We take the first steps towards unifying the fields of database management systems and P2P computing, which so far have received considerable, but separate, attention. We extend database concepts and practice to cover P2P search. Similarly, we extend P2P concepts and practice to support powerful general-purpose query languages such as XQuery and SQL. As a result, we propose the *Unified Peer-to-Peer Database Framework (UPDF)* and corresponding *Peer Database Protocol (PDP)* for general-purpose query support in large heterogeneous distributed systems spanning many administrative domains. They are unified in the sense that they allow to express specific discovery applications for a wide range of data types, node topologies (e.g. ring, tree, graph), query languages (e.g. XQuery, SQL), query response modes (e.g. Routed, Direct and Referral Response), neighbor selection policies (in the form of an XQuery), pipelining characteristics, timeout and other scope options.

The uniformity and wide applicability of our approach is distinguished from related work, which (1) addresses some but not all problems, and (2) does not propose a unified framework.

The results presented in this chapter open four interesting research directions.

*First*, it would be interesting to extend further the unification and extension of concepts from Database Management Systems and P2P computing. For example, one could consider the application of database techniques such as buffer cache maintenance, view materialization, placement and selection as well as query optimization for use in P2P computing. These

techniques would need to be extended in the light of the complexities stemming from autonomous administrative domains, inconsistent and incomplete (soft) state, dynamic and flexible cache freshness policies and, of course, tuple updates. An important problem left open in our work is the question if a query processor can automatically determine whether a correct merge query and unionizer exist, and if so, how to choose them (we require a user to explicitly provide these as parameters). Here approaches from query rewriting for heterogeneous and homogenous relational database systems [37, 73] should prove useful. Further, database resource management and authorization mechanisms might be worthwhile to consider for specific flow control policies per query or per user.

*Second*, it would be interesting to study and specify in more detail specific cache freshness interaction policies between content provider, hyper registry and client (query). Our specification allows expressing a wide range of policies, some of which we outline, but we do not evaluate in detail the merits and drawbacks of any given policy.

*Third*, it would be valuable to rigourously assess, review and compare the Web Service Discovery Architecture (WSDA) and the Open Grid Services Architecture (OGSA) in terms of concepts, design and specifications. A strong goal is to achieve convergence by extracting best-of-breed solutions from both proposals. Future collaborative work could further improve current solutions, for example in terms of simplicity, orthogonality and expressiveness. For practical purposes, our pedagogical service description language (SWSDL) could be mapped to WSDL, taking into account the OGSA proposal. This would allow to use SWSDL as a tool for greatly improved clarity in high-level architecture and design discussions, while at the same time allowing for painstakingly detailed WSDL specifications addressing ambiguity and interoperability concerns.

We are working on a multi-purpose interface for persistent XQueries (i.e. server-side trigger queries), which will roughly correspond to the OGSA publish-subscribe interface, albeit in a more general and powerful manner. The Peer Database Protocol already supports, in a unified manner, all messages and network interactions required for efficient implementations of Peer-to-Peer publish-subscribe and event trigger interfaces (e.g. synchronous pull and asynchronous push, as well as invitations and batching).

*Fourth*, Tim Berners-Lee designed the World Wide Web as a consistent interface to a flexible and changing heterogeneous information space for use by CERN's staff, the High Energy Physics community, and, of course, the world at large. The WWW architecture [74] rests on four simple and orthogonal pillars: URIs as identifiers, HTTP for retrieval of content pointed to by identifiers, MIME for flexible content encoding, and HTML as the primus-inter-pares (MIME) content type. Based on our Dynamic Data Model (DDM), we hope to proceed further towards a self-describing meta content type that retains and wraps all four WWW pillars "as is", yet allows for flexible extensions in terms of identification, retrieval and caching of content. Judicious combination of the four Web pillars, DDM, the Web Service Discovery Architecture (WSDA), the Hyper Registry, the Unified Peer-to-Peer Database Framework (UPDF) and its corresponding Peer Database Protocol (PDP) are used to define how to bootstrap, query and publish to a dynamic and heterogeneous information space maintained by self-describing network interfaces.

***Acknowledgments.*** This chapter is dedicated to Ben Segal, who supported this work with great integrity, enthusiasm and, above all else, in a distinct spirit of humanity and friendship. Gerti Kappel, Erich Schikuta and Bernd Panzer-Steindel patiently advised, suggesting what always turned out to be wise alleys. Testing ideas against the solid background of all member of the EDG WP2 (Grid Data Management) team proved an invaluable recipe in separating wheat from chaff. This work was carried out in the context of a PhD thesis [6] for the European Data Grid project (EDG) at CERN, the European Organization for Nuclear Research, and supported by the Austrian Ministerium für Wissenschaft, Bildung und Kultur.

## REFERENCES

1. Ian Foster, Carl Kesselman, and Steve Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *Int'l. Journal of Supercomputer Applications*, 15(3), 2001.
2. Wolfgang Hoschek, Javier Jaen-Martinez, Asad Samar, Heinz Stockinger, and Kurt Stockinger. Data Management in an International Data Grid Project. In *1st IEEE/ACM Int'l. Workshop on Grid Computing (Grid'2000)*, Bangalore, India, December 2000.
3. Large Hadron Collider Committee. Report of the LHC Computing Review. Technical report, CERN/LHCC/2001-004, April 2001. http://cern.ch/lhc-computing-review-public/Public/Report_final.PDF.
4. E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. *W3C Note 15*, 2001. http://www.w3.org/TR/wsdl.
5. World Wide Web Consortium. Simple Object Access Protocol (SOAP) 1.1. *W3C Note 8*, 2000.
6. Wolfgang Hoschek. *A Unified Peer-to-Peer Database Framework for XQueries over Dynamic Distributed Content and its Application for Scalable Service Discovery*. PhD Thesis, Technical University of Vienna, March 2002.
7. P. Cauldwell, R. Chawla, Vivek Chopra, Gary Damschen, Chris Dix, Tony Hong, Francis Norton, Uche Ogbuji, Glenn Olander, Mark A. Richman, Kristy Saunders, and Zoran Zaev. *Professional XML Web Services*. Wrox Press, 2001.
8. UDDI Consortium. UDDI: Universal Description, Discovery and Integration. http://www.uddi.org.
9. J.D. Ullman. Information integration using logical views. In *Int'l. Conf. on Database Theory (ICDT)*, Delphi, Greece, 1997.
10. Daniela Florescu, Ioana Manolescu, Donald Kossmann, and Florian Xhumari. Agora: Living with XML and Relational. In *Int'l. Conf. on Very Large Data Bases (VLDB)*, Cairo, Egypt, February 2000.
11. A. Tomasic, L. Raschid, and P. Valduriez. Scaling access to heterogeneous data sources with DISCO. *IEEE Transactions on Knowledge and Data Engineering*, 10(5):808–823, 1998.
12. M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 1999.
13. Gnutella Community. Gnutella Protocol Specification v0.4. dss.clip2.com/GnutellaProtocol04.pdf.
14. I. Clarke, O. Sandberg, B. Wiley, and T. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Workshop on Design Issues in Anonymity and Unobservability*, 2000.
15. B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-resilient wide-area location and routing. Technical report, U.C. Berkeley UCB//CSD-01-1141, 2001.
16. I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *ACM SIGCOMM*, 2001.
17. M. van Steen, P. Homburg, and A. Tanenbaum. A wide-area distributed system. *IEEE Concurrency*, 1999.
18. Nelson Minar. Peer-to-Peer is Not Always Decentralized. In *The O'Reilly Peer-to-Peer and Web Services Conference*, Washington, D.C., November 2001.
19. Ann Chervenak, Ewa Deelman, Ian Foster, Leanne Guy, Wolfgang Hoschek, Adriana Iamnitchi, Carl Kesselman, Peter Kunszt, Matei Ripeanu, Bob Schwartzkopf, Heinz Stockinger, Kurt Stockinger, and Brian Tierney. Giggle: A Framework for Constructing Scalable Replica Location Services. In *Proc. of the Int'l. IEEE/ACM Supercomputing Conference (SC 2002)*, Baltimore, USA, November 2002. IEEE Computer Society Press.

20. Leanne Guy, Peter Kunszt, Erwin Laure, Heinz Stockinger, and Kurt Stockinger. Replica Management in Data Grids. Technical report, Global Grid Forum Informational Document, GGF5, Edinburgh, Scotland, July 2002.
21. Heinz Stockinger, Asad Samar, Shahzad Mufzaffar, and Flavia Donno. Grid Data Mirroring Package (GDMP). *Journal of Scientific Programming*, 2002.
22. William Bell, Diana Bosio, Wolfgang Hoschek, Peter Kunszt, Gavin McCance, and Mika Silander. Project Spitfire - Towards Grid Web Service Databases. Technical report, Global Grid Forum Informational Document, GGF5, Edinburgh, Scotland, July 2002.
23. World Wide Web Consortium. XQuery 1.0: An XML Query Language. *W3C Working Draft*, December 2001.
24. T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifiers (URI): Generic Syntax. *IETF RFC 2396*.
25. World Wide Web Consortium. XML-Signature Syntax and Processing. *W3C Recommendation*, February 2002.
26. P. Brittenham. An Overview of the Web Services Inspection Language, 2001. www.ibm.com/developerworks/webservices/library/ws-wsilover.
27. N. Freed and N. Borenstein. Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies. *IETF RFC 2045*, November 1996.
28. World Wide Web Consortium. XML Schema Part 0: Primer. *W3C Recommendation*, May 2001.
29. J. Wang. A survey of web caching schemes for the Internet. *ACM Computer Communication Reviews*, 29(5), October 1999.
30. S. Gullapalli, K. Czajkowski, C. Kesselman, and S. Fitzgerald. The grid notification framework. Technical report, Grid Forum Working Draft GWD-GIS-019, June 2001. http://www.gridforum.org.
31. Marshall Rose. The Blocks Extensible Exchange Protocol Core. *IETF RFC 3080*, March 2001.
32. E. O'Tuathail and M. Rose. Using the Simple Object Access Protocol (SOAP) in Blocks Extensible Exchange Protocol (BEEP). *IETF RFC 3288*, June 2002.
33. International Organization for Standardization (ISO). Information Technology-Database Language SQL. *Standard No. ISO/IEC 9075:1999*, 1999.
34. Matei Ripeanu. Peer-to-Peer Architecture Case Study: Gnutella Network. In *Int'l. Conf. on Peer-to-Peer Computing (P2P2001)*, Linkoping, Sweden, August 2001.
35. Clip2Report. Gnutella: To the Bandwidth Barrier and Beyond. http://www.clip2.com/gnutella.html.
36. Wolfgang Hoschek. A Comparison of Peer-to-Peer Query Response Modes. In *Proc. of the Int'l. Conf. on Parallel and Distributed Computing and Systems (PDCS 2002)*, Cambridge, USA, November 2002.
37. Donald Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys*, September 2000.
38. T. Urhan and M. Franklin. Dynamic Pipeline Scheduling for Improving Interactive Query Performance. *The Very Large Database (VLDB) Journal*, 2001.
39. Jordan Ritter. Why Gnutella Can't Scale. No, Really. http://www.tch.org/gnutella.html.
40. A. Puniyani B. Huberman L. Adamic, R. Lukose. Search in power-law networks. *Phys. Rev*, E(64), 2001.
41. S.E. Deering. *Multicast Routing in a Datagram Internetwork*. PhD Thesis, Stanford University, 1991.
42. Wolfgang Hoschek. A Unified Peer-to-Peer Database Protocol. Technical report, DataGrid-02-TED-0407, April 2002.
43. W. Yeong, T. Howes, and S. Kille. Lightweight Directory Access Protocol. *IETF RFC 1777*, March 1995.
44. Mary Fernandez, Morishima Atsuyuki, Dan Suciu, and Tan Wang-Chiew. Publishing Relational Data in XML: the SilkRoute Approach. *IEEE Data Engineering Bulletin*, 24(2), 2001.
45. Daniela Florescu, Ioana Manolescu, and Donald Kossmann. Answering XML Queries over Heterogeneous Data Sources. In *Int'l. Conf. on Very Large Data Bases (VLDB)*, Roma, Italy, September 2001.
46. Michael Stonebraker, Paul M. Aoki, Witold Litwin, Avi Pfeffer, Adam Sah, Jeff Sidell, Carl Staelin, and Andrew Yu. Mariposa: a wide-area distributed database system. *The Very Large Database (VLDB) Journal*, 5(1), 1996.
47. Ashley Beitz, Mirion Bearman, and Andreas Vogel. Service Location in an Open Distributed Environment. In *Proc. of the Int'l. Workshop on Services in Distributed and Networked Environements*, Whistler, Canada, June 1995.
48. Object Management Group. Trading Object Service. *OMG RPF5 Submission*, May 1996.
49. J. Waldo. The Jini architecture for network-centric computing. *Communications of the ACM*, 42(7), July 1999.
50. Erik Guttman. Service Location Protocol: Automatic Discovery of IP Network Services. *IEEE Internet Computing Journal*, 3(4), 1999.


51. Weibin Zhao, Henning Schulzrinne, and Erik Guttman. mSLP - Mesh Enhanced Service Location Protocol. In *Proc. of the IEEE Int'l. Conf. on Computer Communications and Networks (ICCCN'00)*, Las Vegas, USA, October 2000.
52. Steven E. Czerwinski, Ben Y. Zhao, Todd Hodes, Anthony D. Joseph, and Randy Katz. An Architecture for a Secure Service Discovery Service. In *Fifth Annual Int'l. Conf. on Mobile Computing and Networks (MobiCOM '99)*, Seattle, WA, August 1999.
53. William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. The design and implementation of an intentional naming system. In *Proc. of the Symposium on Operating Systems Principles*, Kiawah Island, USA, December 1999.
54. Bernard Traversat, Mohamed Abdelaziz, Mike Duigou, Jean-Christophe Hugly, Eric Pouyoul, and Bill Yeager. Project JXTA Virtual Network, 2002. White Paper, http://www.jxta.org.
55. Steven Waterhouse. JXTA Search: Distributed Search for Distributed Networks, 2001. White Paper, http://www.jxta.org.
56. Project JXTA. JXTA v1.0 Protocols Specification, 2002. http://spec.jxta.org.
57. Brian Tierney, Ruth Aydt, Dan Gunter, Warren Smith, Valerie Taylor, Rich Wolski, and Martin Swany. A Grid Monitoring Architecture. Technical report, Global Grid Forum Informational Document, January 2002. http://www.gridforum.org.
58. Jason Lee, Dan Gunter, Martin Stoufer, and Brian Tierney. Monitoring Data Archives for Grid Environments. In *Proc. of the Int'l. IEEE/ACM Supercomputing Conference (SC 2002)*, Baltimore, USA, November 2002. IEEE Computer Society Press.
59. Ian Foster, Carl Kesselman, Jeffrey Nick, and Steve Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration, January 2002. http://www.globus.org.
60. Steven Tuecke, Karl Czajkowski, Ian Foster, Jeffrey Frey, Steve Graham, and Carl Kesselman. Grid Service Specification, February 2002. http://www.globus.org.
61. Wolfgang Hoschek. The Web Service Discovery Architecture. In *Proc. of the Int'l. IEEE/ACM Supercomputing Conference (SC 2002)*, Baltimore, USA, November 2002. IEEE Computer Society Press.
62. P. Mockapetris. Domain Names - Implementation and Specification. *IETF RFC 1035*, November 1987.
63. Karl Czajkowski, Steven Fitzgerald, Ian Foster, and Carl Kesselman. Grid Information Services for Distributed Resource Sharing. In *Tenth IEEE Int'l. Symposium on High-Performance Distributed Computing (HPDC-10)*, San Francisco, California, August 2001.
64. IEEE Computer Society. *Data Engineering Bulletin*, 23(2), June 2000.
65. Jayavel Shanmugasundaram, Kristin Tufte, David J. DeWitt, Jeffrey F. Naughton, and David Maier. Architecting a Network Query Engine for Producing Partial Results. In *WebDB 2000*, 2000.
66. Zachary G. Ives, Alon Y. Halevy, and Daniel S. Weld. Integrating Network-Bound XML Data. *IEEE Data Engineering Bulletin*, 24(2), 2001.
67. J. F. Naughton, D. J. DeWitt, D. Maier, A. Aboulnaga, J. Chen, L. Galanis, J. Kang, R. Krishnamurthy, Q. Luo, N. Prakash, R. Ramamurthy, J. Shanmugasundaram, F. Tian, K. Tufte, S. Viglas, Y. Wang, C. Zhang, B. Jackson, A. Gupta, and R. Chen. The Niagara Internet Query System. *IEEE Data Engineering Bulletin*, 24(2), 2001.
68. Annita N. Wilschut and Peter M. G. Apers. Dataflow query execution in a parallel main-memory environment. In *First Int'l. Conf. on Parallel and Distributed Information Systems*, December 1991.
69. Zachary G. Ives, Daniela Florescu, Marc T. Friedman, Alon Y. Levy, and Daniel S. Weld. An adaptive query execution system for data integration. In *ACM SIGMOD Conf. On Management of Data*, 1999.
70. Tolga Urhan and Michael J. Franklin. Xjoin, A reactively-scheduled pipelined join operator. *IEEE Data Engineering Bulletin*, 23(2), June 2000.
71. Beverly Yang and Hector Garcia-Molina. Efficient Search in Peer-to-Peer Networks. In *22nd Int'l. Conf. on Distributed Computing Systems*, Vienna, Austria, July 2002.
72. Adriana Iamnitchi and Ian Foster. On Fully Decentralized Resource Discovery in Grid Environments. In *Int'l. IEEE Workshop on Grid Computing*, Denver, Colorado, November 2001.
73. Y. Papakonstantinou and V. Vassalos. Query rewriting for semistructured data. In *ACM SIGMOD Conf. On Management of Data*, 1999.
74. Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD Thesis, University of California, Irvine, 2000.